Conceptos y Ejercicios de PROGRAMACIÓN

Jorge Humberto Terán Pomier

La Paz - Bolivia

M.Sc.Jorge Humberto Terán Pomier
email:teranj@acm.org
La Paz - Bolivia

Registro de depósito legal
de obras impresas 4-1-551-11

Impresiones Gráficas Apolo
Edición 300 ejemplares
1° Edición - Marzo 2011
La Paz - Bolivia

Prefacio

La programación de computadoras representa un reto, para todos los estudiantes. El problema principal al que se enfrentarán en su vida profesional es el de resolver problemas. Las capacidades que debe lograr un estudiante son el poder enfrentarse a un problema desconocido y encontrar una solución y finalmente traducir ésta solución a un programa de computadora. Su vida profesional requerirá de éstas habilidades permanentemente.

En este texto se ha desarrollado los conceptos en un orden secuencial, introduciendo problemas que gradualmente aumentan su dificultad en base a los conocimientos de capítulos anteriores. A diferencia de los textos clásicos de programación, el énfasis está puesto en los ejercicios que el estudiante debe resolver. Se espera que estos ejercicios les permitan obtener las habilidades requeridas en la programación. Los ejercicios están pensados para que el estudiante vea una serie de problemas diversos que lo lleven a mejorar sus capacidades.

Los problemas presentados están pensados para que se pueda realizar una prueba y comprobar si la solución provee un resultado correcto, verificando diferentes casos de prueba. Este método de verificación es utilizado ampliamente por sistemas de evaluación automáticos. Ayuda al estudiante a detectar sus errores cuando se somete el programa a diferentes casos de prueba, dando una respuesta instantánea sobre los casos de prueba.

Los problemas que se han escogido, algunos han sido desarrollados por el autor y otros son una recopilación de ejercicios de concursos de programación y otros publicados en el internet. El material presentado en este libro puede utilizarse libremente para fines educativos, haciendo mención del autor.

Para el desarrollo del curso no requiere conocimientos avanzados de matemáticas. Solo requiere los conocimientos básicos de geometría, aritmética y álgebra que todo estudiante llevó en el colegio. En los primeros capítulos se plantean los ejercicios en base de estos conceptos porque en opinión del autor es más fácil programar soluciones a problemas de los cuales uno conoce las soluciones.

Los capítulos del 1 al 4 contienen una serie de ejercicios que ayudarán al estudiante a comprender la sintaxis del lenguaje y codificar programas. Desde el capítulo 5 los ejercicios motivan a problemas reales utilizando conceptos del lenguaje de programación. Luego se tienen algunos temas relacionados a los números primos, que nos llevan a pensar en la eficiencia de los programas. Se incluye un capítulo de números de Fibonacci que ayuda al concepto de inicialmente resolver los problemas antes de ponerse a codificar las soluciones. Se da una pequeña introducción a la construcción de métodos y procedimientos. No es el propósito del libro ahondar en la programación orientada a objetos.

Finalmente espero que este texto ayude a docentes y estudiantes a desarrollar las habilidades que se requieren para realizar programas de excelencia.

M.Sc. Jorge Terán Pomier.

Índice general

Capítulo 1

Introducción

Aprender programación no es solamente conocer un lenguaje de programación. También hay que conocer metodologías para resolver problemas en forma eficiente.

En este curso de programación usamos como base el lenguaje Java para el desarrollo de la programación por dos motivos. El primero porque es un lenguaje que hace una verificación fuerte de los programas detectando una gran variedad de errores. Segundo porque es utilizado mayoritariamente en las universidades y colegios para enseñar programación.

Java es un lenguaje orientado a objetos que fue desarrollado por Sun Microsystems. Permite la ejecución de un mismo programa en múltiples sistemas operativos, sin necesidad de recompilar el código. Provee soporte para trabajo en red. Es fácil de utilizar.

1.1. El lenguaje y compilador

Para poder programar en Java es necesario tener un compilador y un entorno de ejecución denominado máquina virtual. Estos pueden descargarse del sitio:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html#need`

Esta versión se denomina Standard Edition En este momento estamos en la versión 1.6. Tambien se conoce como Java Development Kit (JDK).

Existen varios compiladores entre los cuales se pueden mencionar Java Enterprice Edition para desarrollo de aplicaciones empresariales. Java Movil Edition, para desarrollo de aplicaciones para dispositivos móviles, tales como celulares, PDA y otros. Para diferentes tipos de soluciones se puede encontrar compiladores apropiados.

Cuando revise la literatura encontrará textos de denominados Java2 estos se refieren a las versiones posteriores a la 1.2 de Java. En esta versión se incluyen cambios al lenguaje, en los cuales muchas partes fueros reescritas siguiendo la filosofía orientada a objetos.

1.1.1. Instalación de Java

Para instalar el software en el sistema operativo Windows debe descargar el compilador del sitio indicado. Luego de ejecutar el instalador, es necesario definir en las variables de ambiente el camino de acceso al compilador. Esto se hace presionando el botón derecho del mouse en *mi pc* y escogemos

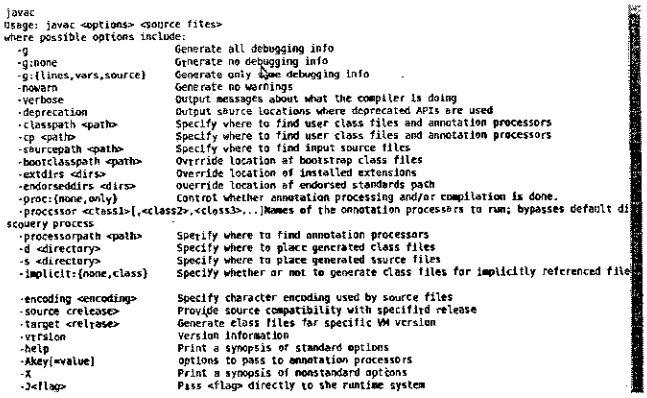


Figura 1.1: Prueba de la instalación

propiedades, luego vamos a la pestaña donde dice *opciones avanzadas*. Damos Clic en Variable de entorno.

En la variable *path* insertamos la dirección donde se encuentra el compilador, podría ser $C : \backslash Java \backslash jdk1{,}6{,}0 \backslash bin;$. Es conveniente colocar al principio de las variables de entorno. Esto para evitar conflictos con otros programas que tenga instalado.

Para probar que todo esté instalado ingresar a *símbolo de sistema* que se encuentra en *menu* y escribir *javac*. La pantalla que obtendrá es similar a la que se muestra en la figura 1.1. La instalación en Linux es más fácil. En las distribuciones basadas en Debian, tales como Ubuntu, el gestor de paquetes ya incluye el compilador Java en su lista de programas. En la línea de comando escribir:

```
sudo apt-get install sun-java6-jdk
```

Luego puede probar con el comando *javac* y obtendrá la misma pantalla que le mostramos con Windows. En linux podemos tener varias versiones instaladas de java. Para ver la versión que estamos usando damos en la línea de comando:

```
update-java-alternatives -l
```

Si tenemos varias versiones instaladas, podemos elegir la que queremos con:

```
sudo update-java-alternatives -s java-6-sun
```

Tanto en linux como en Windows podemos ver la versión del compilador con

```
javac -version
```

1.2. Construir y compilar un programa

Para construir un programa Java es suficiente utilizar el editor de textos del sistema operativo, o uno de su preferencia. Los pasos a seguir son los siguientes:

1. Copiar el programa en el editor de textos.

2. Guardar el mismo con extensión *java*. Tome en cuenta que las letras minúsculas y mayúsculas se consideran diferentes.
3. Compile el programa con el comando *javac NombreDelPrograma.java*
4. Para hacer correr el programa use el comando *java NombreDelPrograma*. Note que hemos obviado la extensión java.

Para ejemplificar el proceso construyamos un programa básico y expliquemos las partes que lo constituyen.

1. Todo en Java está dentro de una clase. Una clase se define

```
public class Nombre {
}
```

2. El código fuente se guarda en un archivo ascii, y debe tener el mismo nombre de la clase y la extensión *.java*.
3. El compilador genera un archivo con extensión *.class* por cada una de las clases definidas en el archivo fuente.
4. Para que un programa se ejecute de forma independiente y autónoma, deben contener el método *main()*.

```
public class Nombre {
   public static void main(String[] args){
       }
}
```

5. Vea que hemos incluido el método *main* en el interior de la clase *Nombre*.
6. La palabra *public* indica que el método es público, vale decir, visible o accesible en toda la clase.
7. La palabra *static* indica que el método está ligado exclusivamente a esta clase.
8. Luego tenemos *void* que significa que no se devuelve ningún valor al programa que lo llamó. En nuestro caso el programa que invoca a este es el sistema operativo.
9. Las instrucciones de nuestro código se ingresan en el método *main()*.
10. Una instrucción para mostrar un texto en pantalla es:

```
System.out.println();
```

11. El texto a mostrar se coloca entre comillas como un parámetro de la instrucción

```
System.out.println("hola");
```

El programa terminado queda como sigue:

```
public class Nombre {
   public static void main(String[] args){
              System.out.println("hola");
       }
 }
```

12. Es muy importante aclarar que las letras en mayúsculas y minúsculas se consideran diferentes.

Una descripción de las partes que constituyen un programa se ven en la figura 1.2.

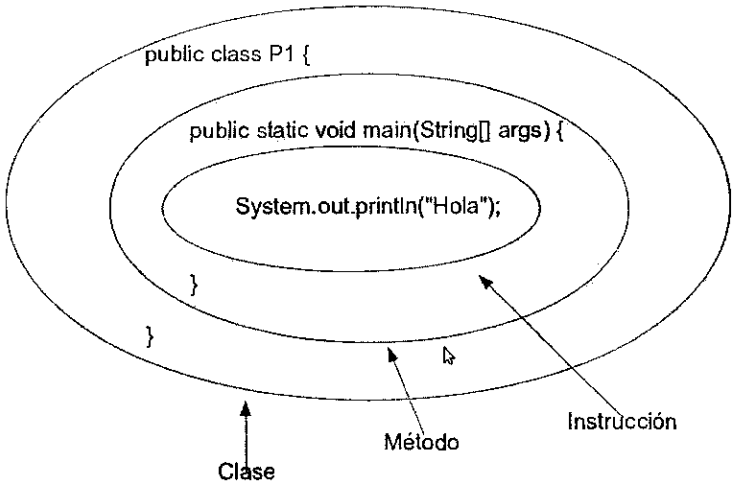


Figura 1.2: Estructura de un programa Java

1.3. Herramientas de desarrollo

Existen una variedad de herramientas de desarrollo integrado (ide) para programas Java. Son ambientes de edición con facilidades para el programador. Las características de una herramienta de desarrollo en relación a un editor de textos convencional son:

1. Más fácil al momento de escribir el programa.
2. Ayuda sobre las diferentes clases y métodos del lenguaje.
3. Depuración de programas en forma sencilla.
4. Es más fácil probar los programas.
5. Es más fácil el imponer normas institucionales.
6. Menos tiempo y esfuerzo.
7. Administración del proyecto.

Para el desarrollo elegimos la herramienta *Eclipse*. Las razones para esto son las siguientes:

1. Es gratuito y viene con una licencia GPL. se puede descargar de

   `http://www.eclipse.org/downloads/`

   Para utilizar el ambiente para Java debe descargar *Eclipse Classic*, de 32 bits o 64 bits de acuerdo al sistema operativo y hardware que tenga.

2. Eclipse es una plataforma completa de desarrollo. En el curso solo utilizaremos algunas funcionalidades, sin embargo, es deseable comenzar aprendiendo un entorno de desarrollo profesional.

3. Eclipse tiene muchos complementos que fácilmente ayudan a extender el ambiente de desarrollo a otros lenguajes. Se puede trabajar con *HTML*, *PHP*, *UML*, etc.

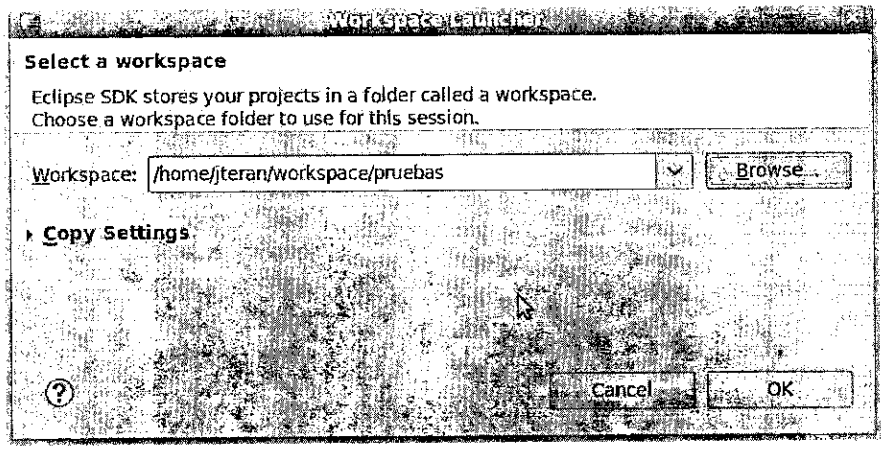


Figura 1.3: Especificar el área de trabajo en Eclipse

4. Extensas ayudas para aprender JAVA.

5. Fácil de aprender:

6. Puede funcionar tanto en Linux o Windows.

7. Esta desarrollado en Java.

1.3.1. Instalación de Eclipse

La instalación de Eclipse es muy simple sea su ambiente de desarrollo Linux o Windows, descargue la herramienta y copie a un directorio de su disco. Luego ejecute Eclipse. El único requisito es que el java esté instalado previamente y accesible desde la línea de comando.

1.3.2. Construir y hacer correr un programa

El proceso de construir un programa utilizando Eclipse es sencillo. Hay que tener en cuenta una serie de conceptos que se explican a continuación:

1. Una vez iniciado Eclipse obtendremos la pantalla que se muestra en la figura 1.3. El *workspace* es el área de trabajo. Esto significa el directorio donde se crearan sus proyectos y programas. Nos muestra la ruta del directorio donde está. A esta pantalla podemos llegar también con la opción *switch workspace* que se encuentra en el menú *file*. Si desea hacer una copia de respaldo de todos sus proyectos es suficiente copiar este directorio.

2. Cuando se inicia por primera vez Eclipse se presentará una pantalla de bienvenida que provee información y ayuda para su uso (figura 1.4). Para salir de esta pantalla se marca la X que está al lado de la pestaña que dice *Welcome*. Esta pantalla aparecerá solo la primera vez.

3. Cerrando la pantalla de bienvenida se obtiene el ambiente de trabajo que se muestra en la figura 1.5. Esta pantalla tiene varias partes:

   - La ventana que dice *Package*. Esta sección nos muestra los proyectos y programas que vamos creando.
   - La ventana *Outline* que nos muestra los métodos de nuestro programa.

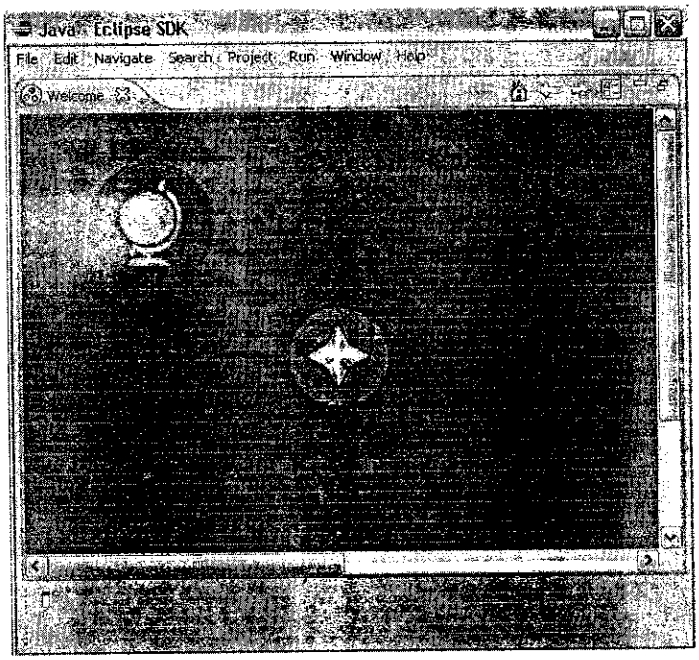


Figura 1.4: Pantalla de bienvenida de Eclipse

- La parte inferior cuando ejecutemos el programa nos mostrará la consola. También se muestra la documentación *Javadoc*, *Problemas* y *declaraciones*.
- Al centro se ve un espacio donde se escribirá el código del programa.

4. Para empezar a crear un programa el primer paso es crear un proyecto. Dentro del proyecto estarán todos los programas del mismo. Desde la pestaña *File* escogemos *new* → *java Project* y obtenemos la pantalla de la figura 1.6. Ingresamos el nombre del proyecto y escogemos *Finish*. En este momento podemos cambiar varios aspectos del proyecto, como ser, la versión de java que se utilizará. Un aspecto muy importante que hay que resaltar que si crea un proyecto que no es de java no podrá ejecutar un programa java.

5. Una vez que ha creado un proyecto, es necesario crear un archivo que contendrá su programa. En eclipse se denomina crear una clase. Para esto colocamos el cursor de ratón en el nombre del proyecto con el botón derecho del ratón entramos al menú. Escogemos *new* → *class*. Obteniendo la pantalla de la figura 1.7. Aquí ingresamos el nombre del programa y escogemos la opción *public static void main(String[] args)* para indicar que es un programa ejecutable. Obteniendo la pantalla de la figura 1.8

6. En esta pantalla ya vemos un programa con una *clase* que tiene el mismo nombre del archivo, y un método principal que es donde se introducirán las instrucciones. Las líneas

```
/**
 * @param args
 */
```

Es donde colocamos los comentarios generales del programa, tales como parámetros, fecha, autor. Todos los bloques de comentarios (múltiples líneas) comienzan con /* y terminan con */.

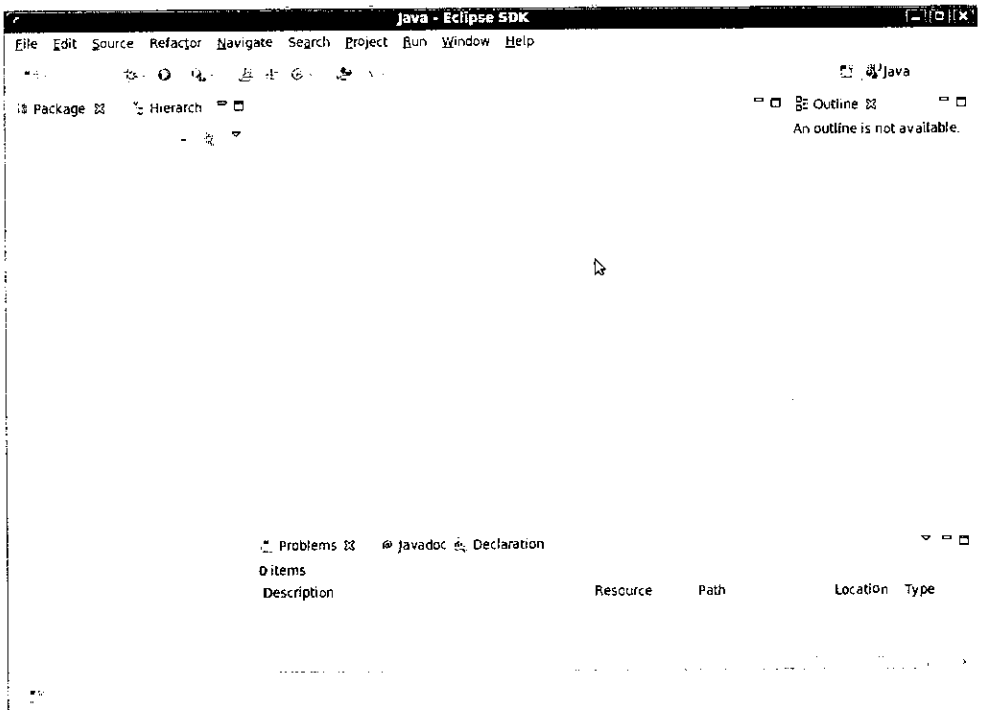


Figura 1.5: Entorno de trabajo de Eclipse

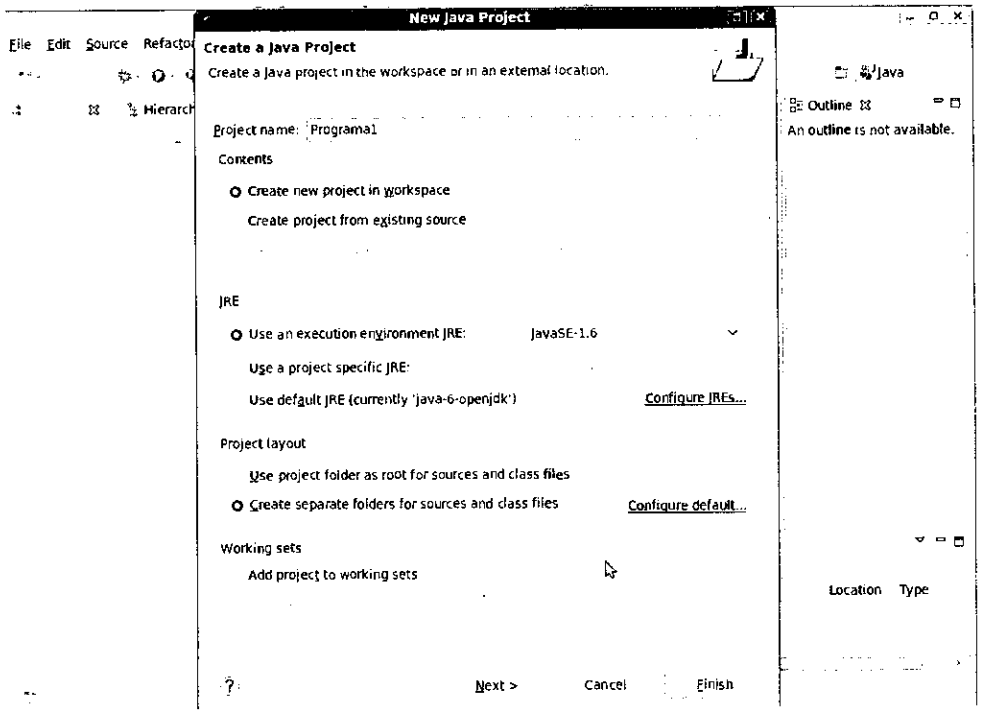


Figura 1.6: Opciones para crear un proyecto

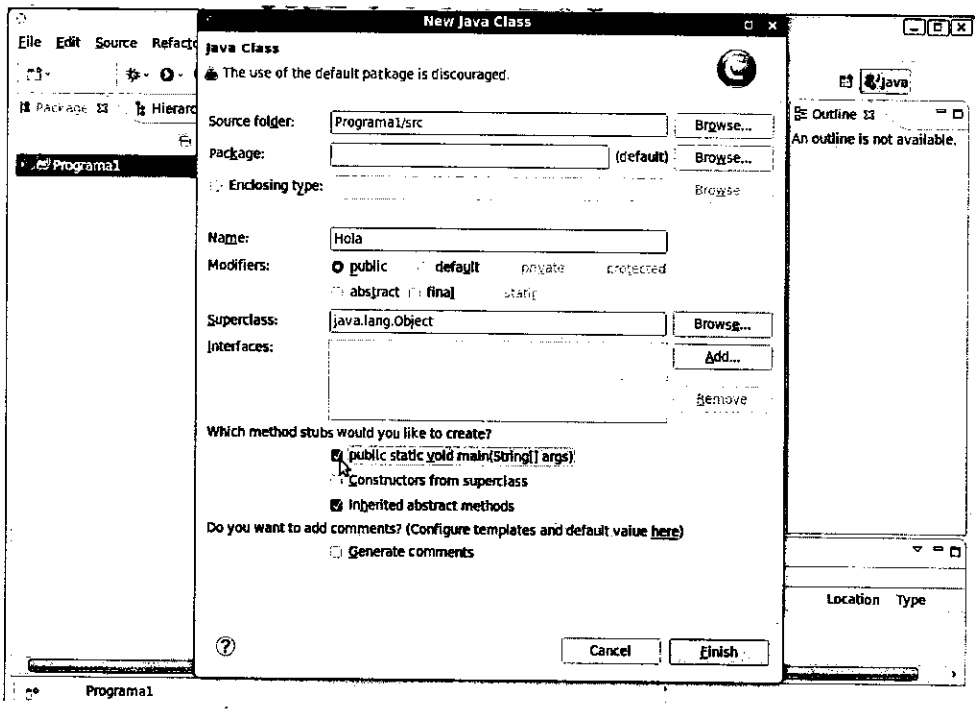


Figura 1.7: Opciones para crear una clase

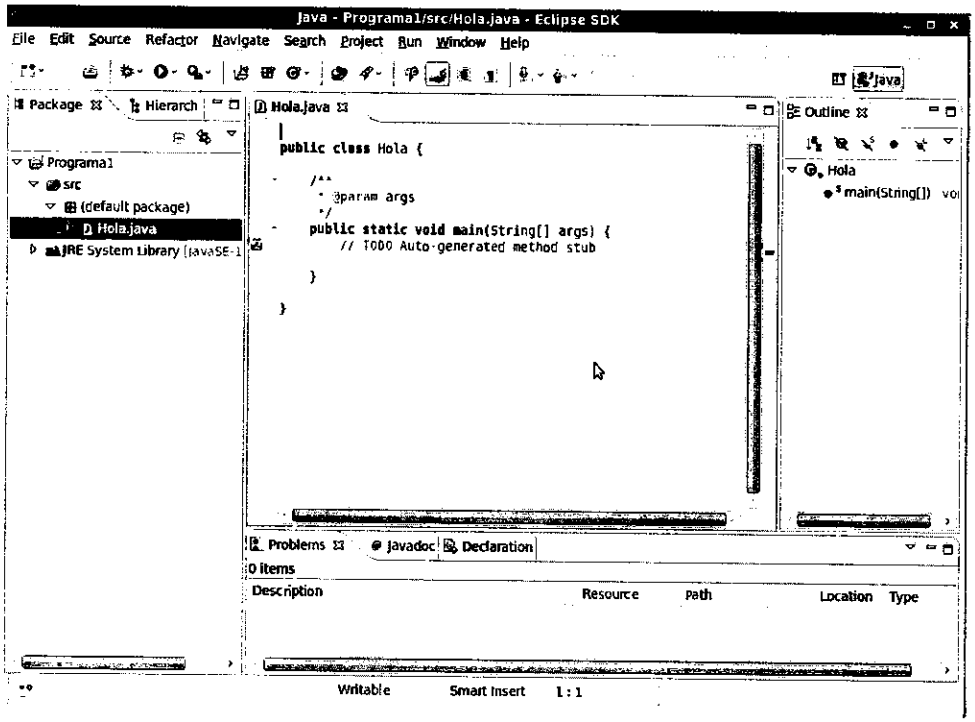


Figura 1.8: Plantilla de programa inicial

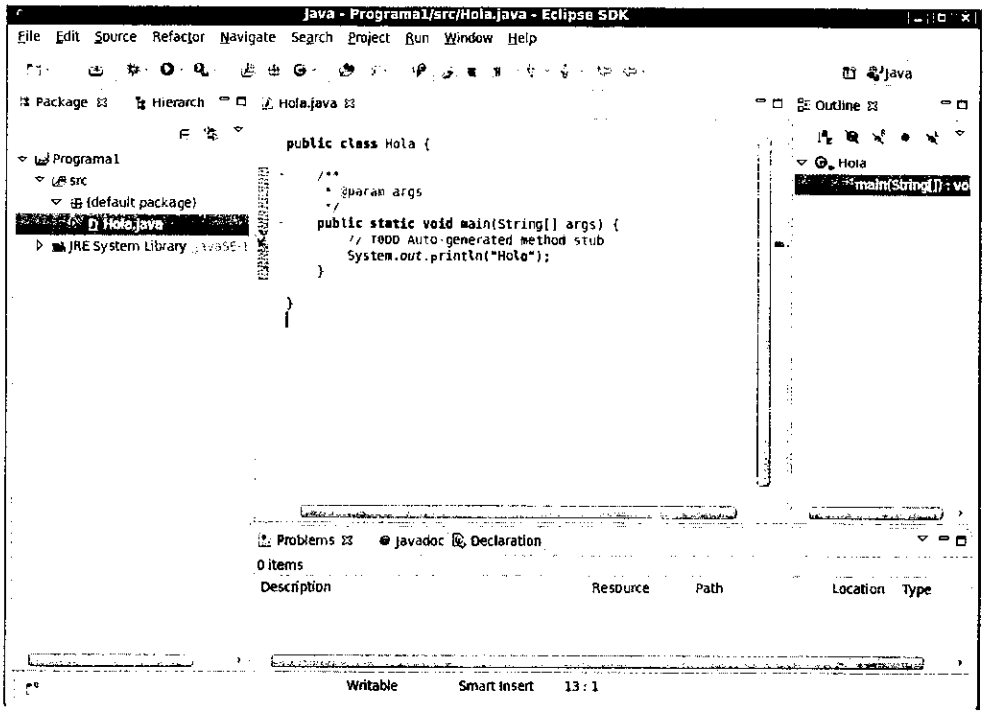


Figura 1.9: Programa para mostrar un texto

Los comentarios que comienzan con @ son instrucciones que el programa *javadoc* utiliza para generar la documentación. En este caso *@param args* indica que se pueden pasar valores desde la línea de comando. Esto no se procesa solo sirve para documentar.

7. Cuando un comentario se pone en una sola línea, no es un conjunto de líneas, se utiliza //. El comentario *// TODO Auto-generated method stub* solo dice que es un comentario auto generado. Aquí describimos que hace el programa.

8. A continuación podemos escribir las instrucciones que se ejecutarán. En el ejemplo pusimos

   ```
   System.out.println("Hola");
   ```

   que especifica que queremos mostrar la palabra "Hola"por pantalla.

9. Antes de ejecutar el programa vea el mismo finalizado (figura 1.9). Para ejecutar el mismo, con el botón derecho del ratón en el nombre del programa escogemos *Run as* → *Java Aplication*. Ahora veremos la salida del programa en una ventana que se llama *Console*.

   Si en algún caso no tenemos la consola visible vamos a *Window* → *Show view* → *Console*.

1.3.3. Estructura de directorios

Cuando creamos un proyecto y un programa java obtenemos la siguiente estructura de directorios:

- En el área de trabajo (workspace) una carpeta con el nombre del proyecto.
- En el interior de la carpeta con el nombre de proyecto dos carpetas *bin* y *src*.

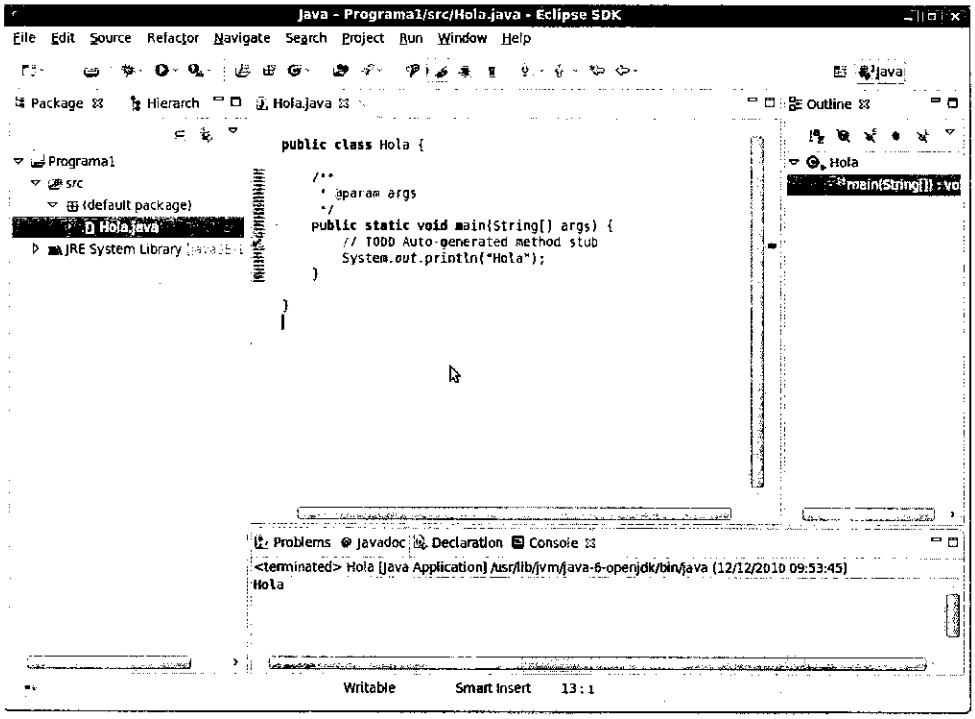


Figura 1.10: Salida del programa

- En la carpeta *bin* los programas compilados con la extensión *.class*.
- En la carpeta *src* los programas fuentes que tienen la extensión *.java*.

Al recorrer las carpetas con el explorador de proyecto podremos ver esta estructura, así como, recorriendo las carpetas desde la línea de comando.

1.4. Ejercicios

1. Instalar el compilador Java.
2. Instalar el Editor Eclipse.
3. Escribir un programa que muestre su nombre utilizando Eclipse.
4. Busque en qué lugar está el directorio *workspace*.
5. Describa los directorios creados en el proyecto.
6. Compile y haga correr el programa desde la línea de comando.
7. Compile y haga correr el programa desde Eclipse.

Capítulo 2

Tipos de Datos

2.1. Introducción

Seguramente usted ha utilizado una computadora para realizar una serie de tareas. Y son muy buenas para las tareas repetitivas. Puede realizar las mismas una y otra vez sin cansarse.

Un equipo debe ser programado para realizar tareas. Diferentes tareas requieren diferentes programas. Por ejemplo para que se pueda escribir un programa primero ha tenido que construirse un editor que nos permite introducir el código y guardarlo en un archivo.

Las computadoras solo ejecutan un conjunto de operaciones básicas muy rápidamente. Con este conjunto básico es que se construyeron los programas que nos permiten posteriormente hacer tareas más complejas. Las instrucciones básicas pueden obtener datos de un lugar de la memoria, sumar dos números, guardar en la memoria, si el valor es negativo continuar en otra instrucción.

Para construir los programas que se usan hoy, tal es el caso de Java, en base de estas instrucciones básicas se crearon lenguajes con muchas más instrucciones para hacer la tarea de programación más sencilla.

Un programa de una computadora es una secuencia de instrucciones necesarias para realizar una tarea.

2.2. Entender la actividad de la programación

La actividad de la programación consiste en escribir algoritmos para resolver problemas o tareas. Por esto es necesario definir con precisión que entendemos por algoritmo.

Un algoritmo es una secuencia de pasos que tiene un inicio y un final. Quiere decir que finaliza en algún momento. Los pasos deben ser precisos.

Por ejemplo si queremos guardar en B la suma de $A + 2$ el programa podría ser como sigue:

1. Condiciones iniciales A tiene un valor,el valor de B no nos interesa-
2. Obtener el valor de A.
3. Sumar 2. -
4. Guardar el resultado en B.

5. Condiciones finales B contiene el valor de $A + 2$.

Analizando el ejemplo podemos ver que las instrucciones son precisas y el orden en el que se ejecutan es importante. Si cambiamos el orden seguro que el resultado será diferente.

En los algoritmos no se permiten valores no cuantificados claramente. Por ejemplo un algoritmo para realizar una receta de cocina podría ser:

1. Poner 1 taza de harina.
2. Agregar una taza de leche.
3. Agregar un huevo.
4. Poner una cucharilla de sal.
5. Mezclar.
6. Hornear a 220 grados.

En este ejemplo la secuencia es precisa. Si por ejemplo si especificamos sal al gusto, deja de ser un algoritmo dado que el concepto *al gusto* no es un valor preciso.

Otro caso en que una secuencia deja de ser un algoritmo es cuando después de una cantidad de pasos no termina

1. Inicio.
2. Asignar a N el valor de 100.
3. Repetir hasta que N mayor que 256.
4. Asignar a N el valor $N/2$.
5. Fin de repetir.
6. Fin del algoritmo.

Claramente se ve que el algoritmo no termina puesto que cada vez vamos reduciendo el valor de N en lugar de que vaya creciendo. Aún cuando se ha definido un inicio y un final, el algoritmo no termina, estará ejecutando continuamente.

Como dijimos los algoritmos son secuencia de pasos, no importando el lenguaje que se utiliza, o como se expresó el mismo. En el curso expresaremos los algoritmos en lenguaje Java. Muchas veces usamos una mezcla de lenguaje español con Java para explicar una tarea, esto se denomina pseudo lenguaje.

2.3. Reconocer errores de sintaxis y de lógica

Cuando escribimos un programa existen dos tipos de errores, los de sintaxis y los de lógica. Los errores de sintaxis son aquellos en los que existen errores en la construcción de las instrucciones. Por ejemplo: la carencia de un punto y coma al final de una instrucción, una palabra que debe comenzar con mayúsculas y se escribió con minúsculas, un error ortográfico, etc.

Este tipo de errores es detectado por el compilador cuando compilamos el programa. El entorno de desarrollo Eclipse lo marca como un error ortográfico. Para ejecutar un programa no pueden existir errores de este tipo.

Los errores de lógica son más difíciles de descubrir dado que el compilador no puede detectarlos. Estos errores se producen cuando el programa no hace lo que deseamos. Por ejemplo supongamos que queremos contar el número de números que hay en una secuencia y hemos realizado la suma de los números, claramente es un error de lógica.

2.4. Tipos de datos

Cada valor que utilizamos en Java tiene un tipo. Por ejemplo *"Hola"* es de tipo cadena, un número puede ser de tipo entero. Otro ejemplo es *System.out* que tiene un tipo *PrintStream,*.

¿Como definimos datos en un programa? Para definir un dato la sintaxis que se utiliza en Java es:

```
Nombretipo nombreVariable = valor;
```

Los nombres se construyen bajo las siguientes restricciones:

1. Comienzan con una letra mayúscula o minúscula.
2. Pueden contener letras y números.
3. También puede incluir el símbolo guión bajo (_) o el símbolo dolar ($).
4. No se permiten espacios.

Algunos nombres válidos son *nombre*, *Caso_1*, *cedulaIdentidad*. Por convención de los nombres deben comenzar con una letra minúscula.

El signo = se denomina operador de asignación y se utiliza para cambiar el valor de una variable. En el ejemplo *nombreVariable* cambia su contenido con el contenido de valor.

Hay dos tipos de datos, los del núcleo del lenguaje denominados tipos primitivos o básicos y los implementados en clases. Los tipos básicos son:

Tipo	Descripción	Tamaño
int	Números enteros en el rango 2,147,483,648 - 2,147,483,647	4 bytes
byte	Descripción de un número entre -128 - 127	1 byte
short	Un entero en el rango de -32768 - 32767	2 bytes
long	Un entero en el rango de -9,223,372,036,854,775,808 9,223,372,036,854,775,807	8 bytes
double	Número de precisión doble de punto flotante en el rango de $\pm 10^{308}$ con 15 dígitos decimales	8 bytes
float	Número de precisión simple de punto flotante en el rango de $\pm 10^{38}$ con 7 dígitos decimales	4 bytes
char	Caracteres en formato Unicode	2 bytes
boolean	Un valor que representa verdadero o falso	1 bit

Los caracteres de la tabla Ascii normalmente se representan en un byte, sin embargo, para poder representar caracteres en diferentes lenguajes (español, inglés, árabe, etc.), se utiliza una norma denominada *Unicode*.

Un byte representa 8 bits o sea 8 dígitos binarios. Un número entero de 4 bytes utiliza un bit para el signo y 31 para almacenar un número. De esta consideración se deduce que el numero mas grande que se puede almacenar es $2^{31} - 1 = 2,147,483,647$.

Para definir los valores numéricos podemos utilizar una de las siguientes sintaxis.

- tipo nombre;
- tipo nombre = valor;

Vea que cada instrucción termina con un punto y coma. Por ejemplo:

```
int i;
```

Esto permite definir una variable de nombre *i* de tipo entero que no tiene un valor inicial. En este caso el valor inicial es *null*;

Si definimos:

```
int i=3;
```

Significa que estamos definiendo una variable de nombre *i* con valor inicial 3

En el caso de las variables de punto flotante es necesario poner un punto decimal para indicar que los valores son del tipo con decimales. Por ejemplo:

```
double f=10.0;
```

No podemos colocar solo 10 porque este es un número entero y es necesario que los tipos igualen en toda asignación.

Errores de desborde

Cuando realizamos una operación con una variable, y excedemos el valor máximo que podemos almacenar, se produce un desborde. Supongamos que en una variable de tipo short tenemos almacenado el número 32767. Si agregamos 1 a ésta, no obtendremos 32768, el resultado es -32768 porque hemos excedido el número máximo que podemos almacenar en este tipo de variable.

Para entender ésto, vemos que pasa, convirtiendo ésto a números binarios. Si al valor 32767 = 01111111, sumamos uno la respuesta es 10000000 que en la representación de la computadora equivale a -32768

El desborde lo identificamos cuando al realizar una operación el resultado es incorrecto. Por ejemplo si esperamos un resultado positivo y obtenemos un resultado negativo.

Errores de redondeo

Cuando representamos un número en notación de punto flotante, el resultado se expresa de la forma *enteros.decimalesEexponente*. Por ejemplo 5,6666664$E7$ significa 5,6666664$x10^7$. Con esta representación las conversiones no son exactas. Por ejemplo 10/3 es 3,3333333333333333 sin embargo en la computadora nos da 3,3333333333333335. Esto podemos probar con el siguiente código:

```
double a = 10.0/3;
System.out.println(a);
```

Debido a estos errores que se producen hay que tener mucho cuidado como se manejan los números de punto flotantes para no obtener resultados erróneos. Esto es principalmente crítico en cálculos financieros donde no se permiten errores por la precisión de la representación de los números en la computadora.

2.4.1. Ubicación en la memoria de la computadora

Todos las variables se almacenan en la memoria de la computadora en forma secuencial. Esto quieres decir uno a continuación del otro.

Cuando tenemos una definición, por ejemplo, *int* $a = 1234$; el contenido de la variable a es 1234. El nombre a representa la dirección de memoria donde está ubicado. En este caso donde comienzan los 4 bytes de la variable a.

Figura 2.1: Direccionamiento de las variables en la memoria

En la figura 2.1 podemos ver que el nombre de una variable básica es un apuntador a su contenido, que se almacena en la cantidad de bytes que corresponde al tipo de la variable.

Cuando el tipo no corresponde a uno básico, el nombre de la variable no apunta al contenido. Consideremos como ejemplo una cadena. Una cadena de caracteres puede tener cualquier longitud. Por esto no es posible, que el nombre apunte al contenido. Lo que se hace es que el nombre apunte a un lugar, donde se encuentra la dirección de memoria donde está la cadena.

- Lo que decimos es que la variable es un puntero a la dirección de memoria donde está el contenido. Si hacemos una operación con ésta variable, por ejemplo sumar 1, se estaría cambiando la dirección del contenido. Para evitar ésto todas las operación sobre variables que no corresponden a los tipos básico se realizan con métodos.

2.4.2. Variables y constantes

Se pueden definir dos tipos de datos, los *variables* y los *constantes*. **Variables** son aquellos que pueden cambiar su valor durante la ejecución del programa. **Constantes** son aquellos que no pueden cambiar su valor durante la ejecución del programa.

Las variables se definen como vimos en la definición de variables. Simplemente se coloca el tipo y el nombre de la misma.

Para definir un valor constante por ejemplo

```
double PI=3.1416;
```

puede realizarse como una variable normal. Esta definición no permitiría evitar que cambiemos el valor de PI. Si alteramos el valor por un error de lógica será difícil de hallar el mismo. Para especificar al compilador que no se puede modificar el valor utilizamos la palabra *final* en la definición quedando:

```
double PI=3.1416;
```

Ahora, si tratamos de modificar esta variable obtendremos un error de compilación.

2.4.3. Tipos de datos implementados en clases

Para los tipos de datos básicos existen clases que aportan una variedad de métodos para muchas tareas habituales, esta clases son:

Tipo	Descripción
Integer	Clase que define números de tipo *int*
Byte	Clase que define números de tipo *byte*
Short	Clase para definir datos de tipo *short*
Long	Clase para definir datos de tipo *long*
Double	Clase para definir datos de tipo *double*
Float	Clase para definir datos de tipo *float*
Boolean	Clase para definir datos de tipo *boolean*

Vea que el tipo comienza con letra mayúscula. Por convención todas las clases comienzan con un letra mayúscula.

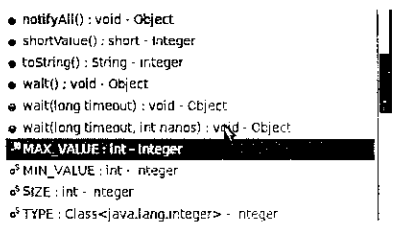
El propósito de estos tipos como dijimos es acceder a sus métodos. Para acceder a un método se coloca el nombre de la variable un punto y el nombre del método. Definamos una variable de tipo *Integer*

```
Integer entero=5;
```

Ahora si se desea conocer cuantos bits tiene la variable *entero* utilizamos el método *SIZE*. El código siguiente nos permite comprobar que el número de bits de una variable de tipo entero, es 32.

```
Integer entero=5;
System.out.println(i.SIZE);
```

Cada clase usada para definir datos tiene sus propios métodos. Para ver la lista de métodos utilizando el editor Eclipse escribimos el nombre de la variable, punto, y obtenemos una lista de métodos de los cuales podemos escoger el que necesitamos. Una parte de la ayuda para *Integer* se muestra:



Entre los métodos más comunes tenemos:

Tipo	Descripción
toString()	Convertir un entero a una cadena
MAX_VALUE	El valor máximo que puede almacenar
MIN_VALUE	El valor mínimo que puede almacenar
reverse(*int value*)	Invertir los bits del número
toBinaryString(*int value*)	Convertir a una cadena en binario
toHexString(*int value*)	Convertir a una cadena en hexadecimal
toOctalString(*int value*)	Convertir a una cadena en octal

Es necesario aclarar algunos aspectos sobre como han sido construidos los nombres de los métodos de acuerdo a las convenciones del Java:

1. Los métodos comienzan con una letra minúscula
2. Cuando tienen más de dos nombres el segundo nombre comienza con una letra mayúscula, por ejemplo *toString()*
3. Los paréntesis significan que es un método y puede o no tener parámetros. Por ejemplo *toString()* no tiene parámetros. En cambio toBinaryString(*int value*) recibe un parámetro que es un valor entero y devuelve su equivalente en binario. Lo que especifica el método, es que el parámetro es un valor de tipo *int*;
4. Cuando el nombre está todo en letras mayúsculas significa que es un valor constante.

Por ejemplo si queremos mostrar el equivalente del número 12345 en binario el código sería el siguiente:

```
System.out.println(Integer.toBinaryString(12345));
```

2.5. Caracteres

Como habrá notado no existe una clase *Char*, el tipo de datos es *char* y no tiene métodos. Los caracteres corresponden a los caracteres de la tabla ascii. Mostramos algunos de los caracteres de la tabla:

	0	1	2	3	4	5	6	7	8	9
4	(	)	*	+	,	-	.		0	1
5	2	3	4	5	6	7	8	9	:	;
6	¡	=	¿	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	/	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	

Los caracteres están acomodados correlativamente, si vemos el carácter 0 es número 48 el carácter 1 es el 49 y así sucesivamente. Si vemos las letras mayúsculas comienzan en el 65 hasta el 90.

Para definir un carácter tenemos dos opciones:

- Asignamos a la variable el valor numérico del caracter. Por ejemplo para definir la letra *A* podemos escribir

```
char letraA=65;
```

- si no conocemos el valor ascii del carácter podemos escribir el mismo entre apostrofes. Para definir la misma letra *A* escribimos

```
char letraA='A';
```

2.5.1. Interpretación de los datos

Cuando escribimos un número éste puede representar tanto un carácter como un número. ¿Cómo sabe el compilador si es un número o un caracter?

El compilador no puede determinar esto. Uno debe decir que es lo que representa. Para esto se utiliza un concepto que en Java se denomina *cast*. La sintaxis consiste en colocar el tipo entre paréntesis delante de la variable.

Por ejemplo si queremos asignar el caracter 80 de una variable entera a una variable caracter debemos hacer un *cast*. En una asignación ambos lados deben ser del mismo tipo. Un ejemplo seria:

```
int i=80;
char c=(char)i;
```

Lo mismo ocurre con todos los tipos de variables, por ejemplo para asignar una variable *int* a una *long*

```
int i=80;
long l=(long)i;
```

2.5.2. Salida por pantalla

Para mostrar los datos por pantalla se utiliza la clase *System.out* con el método *print*. Esta clase solo puede mostrar secuencias de caracteres en la pantalla. Cuando imprimimos un número primero se convierte en una cadena de caracteres y luego se imprime. Por ejemplo si tenemos *int* $i = 97$ y deseamos imprimir el valor de i primero se debe imprimir el carácter 9 luego el 7 para que se muestre el número 97. Si se envía directamente el 97 se mostrará por pantalla la *a*.

El código es el siguiente:

```
int i=97;
System.out.print(i);
```

Ahora, si queremos que en lugar de 97 se imprima la letra *a* hacemos un *cast*, el código es el siguiente:

```
int i=97;
System.out.print((char)i);
```

Para imprimir un texto, se coloca el mismo entre comillas.

```
System.out.print("hola");
```

Si se quiere imprimir textos, y números debe concatenar ambos. Esto se hace con el símbolo +. Por ejemplo

```
int i=123;
System.out.print("El valor de i es "+i);
```

En el ejemplo tenemos una cadena y luego i es convertido a cadena con el método *toString()*, dando un resultado que es una cadena. Si codifica

```
int i=123;
System.out.print(i+" es el valor de i");
```

Se producirá un error de sintaxis. Solo se pueden concatenar cadenas. La instrucción trata de concatenar un entero con una cadena. Para eliminar el problema colocaremos una cadena al principio. Esta cadena de longitud cero, porque no encierra nada, se denomina cadena de valor *null*. La forma correcta de codificar es:

```
int i=123;
System.out.print(""+i+" es el valor de i");
```

Caracteres especiales

Existen algunos caracteres especiales que son:

Carácter	Descripción
\t	El carácter de tabulación, que se denomina *tab*. En la tabla ascii es el carácter 9
\r	Retorno de carro, Significa volver al principio de la línea, es el carácter 13. Este lo denominamos *cr*
\f	Avance de línea, es el carácter 12, denominado lf

Cuando usamos *System.out.print* se muestran los resultados, se muestran en una línea de la pantalla pero no se avanza a la siguiente línea. Si queremos que los caracteres siguientes contienen en la próxima línea debemos indicar esto con los caracteres especiales.

Dependiendo del sistema operativo hay diferencias. En Windows de utiliza cr y lf para avanzar una línea y regresar al principio. En Linux es suficiente lf. En Mac cr. Para evitar este problema podemos utilizar la instrucción *System.out.println*. El método *println* indica que una vez completada la instrucción se imprima los caracteres necesarios para avanzar una línea e ir al comienzo.

Si deseamos incluir estos caracteres en una cadena solo lo incluimos en medio del texto. La instrucción *System.out.print("que \r dice")* hará que se imprima *que* en una línea y *dice* al principio de la siguiente línea.

Como vimos el carácter \indica que a continuación hay un caracter de control. Si queremos imprimir el carácter \entonces debemos colocar dos \\.

2.5.3. Despliegue de números con formato

Consideremos el siguiente código:

```
double total = 35.50;
System.out.println("Total ="+total);
```

Al ver la respuesta vemos que el resultado es $Total = 35{,}5$. Bien aún cuando representa el mismo número la salida presenta un solo decimal. El formato de salida elimina los ceros de la izquierda y los de la derecha después del punto decimal.

Para imprimir valores numéricos con un formato específico se utiliza la instrucción *System.out.printf()* que tiene la siguiente sintaxis:

```
System.out.printf("formato", variable);
```

En formato se escriben los textos y las características de formato de la variable que queremos imprimir. Por ejemplo para imprimir *Total = 35.50* en el campo de formato escribimos "Total%5.2f". Esto significa que imprimiremos la palabra *Total* seguida de un espacio luego viene una variable de punto flotante de tamaño fijo de 5 caracteres de los cuales 2 son decimales. Para el ejemplo la instrucción es:

```
System.out.printf("Total \%5.2f",total);
```

Los formatos más comunes se muestran en la siguiente tabla:

Código	**Descripción**	**Ejemplo**
d	Decimal	123
x	Entero hexadecimal	4B
o	Entero octal	12
f	Punto flotante	35.50
e	Punto flotante con exponente	1.25e+2
s	Una cadena	Hola
n	Fin de línea independiente del sistema operativo	
-	Alineación a la izquierda	1.23 seguido de espacios
0	Mostrar los ceros de la izquierda	0012
+	Mostrar el signo + en los números	+123
(	Los números negativos se muestran entre paréntesis	(123)
,	Mostrar separadores de miles	1,234

Veamos unos ejemplos:

Formato	**Salida**
System.out.printf(" %x",1234)	4d2
System.out.printf(" %6.2f%6.2f",12.12,25.45")	12,12 25,45
System.out.printf(" %+d",-1234)	-1234
System.out.printf(" %(d",-1234)	(1234)
System.out.printf(" %,d",1234)	1,234
System.out.printf(" %06d",1234)	001234

2.6. Ejercicios

1. Problemas de tipos de datos y caracteres
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   a) Los siguientes ejercicios tienen la finalidad de que conozca los caracteres y los pueda encontrar en el teclado. Solo debe utilizar la instrucción *System.out.print()* y *System.out.println()*. Lo que el programa debe realizar es imprimir en pantalla las salidas que se muestran en cada uno de los incisos.

   1)
   ```
   +-----------+
   | Su nombre |
   +-----------+
   ```

   2)
   ```
   -------
     -_-
   -------
   ```

   3)
   ```
   <------->
   <_______>
   <%%%%%%%>
   </////////>
   ```

   4)
   ```
   +--------+
   +        +
   +--------+
   +        +
   +--------+
   +        +
   +--------+
   +        +
   +--------+
   ```

   5)
   ```
      |      |
      |      |
   ---------------
      |      |
      |      |
   ```

   b) En los siguientes incisos utilice la instrucción *System.out.println()* para cada enunciado.

   1) Muestre el código ascii de los caracteres: 0,A,a,Ñ,ñ.
   2) Muestre el código ascii de todas las vocales con acento.
   3) Muestre el carácter correspondiente a los enteros: 49, 66,64,97.
   4) Muestre la representación binaria de las letras: A,a y describa que bits son diferentes.
   5) Escriba una instrucción que muestre el resultado de sumar uno a los caracteres: 0,A,a,Ñ,ñ.

   c) Escriba una instrucción *System.out.printf()* que produzca el resultado mostrado.

   1) El resultado de 1/3 con 2 dos enteros y 3 decimales
   2) El valor máximo una variable entera con separadores de miles.
   3) Tres números enteros de dos dígitos, cada uno ocupando un espacio de 5 caracteres.
   4) Escriba un número flotante, y un número entero separados por un espacio.

   d) Considere la siguiente descripción de datos

```
double  a1 = 123.77,
a2 = 425.23,
a3 = 319.44,
a4 = 395.55;
```

Si deseamos imprimir el 10 % de cada uno de los valores el código es:

```
System.out.printf("\%f\%n",a1/10);
System.out.printf("\%f\%n",a2/10);
System.out.printf("\%f\%n",a3/10);
System.out.printf("\%f\%n",a4/10);
```

Para imprimir el 10 % de la suma de los valores escribimos:

```
System.out.printf("\%f\%n",(a1+a2+a3+a4)/10);
```

La suma de los porcentajes individuales no iguala con el porcentaje de la suma ¿Por que? Busque una solución para que la suma de los valores individuales iguale con la suma total.

Capítulo 3

Operadores aritméticos y lectura de teclado

3.1. Introducción

En este capítulo se explica como trabajar con los valores enteros utilizando operaciones tanto en bits individuales como en números. Como convertir de un tipo de datos a otro y finalmente como podemos ingresar los mismos por teclado.

3.2. Trabajando en binario

Cuando sea posible es preferible trabajar en binario, dado que, las computadoras trabajan en números binarios y esto mejorará el tiempo de proceso. Los operadores disponibles para trabajar manejo de bits son:

Operador	Descripción	Ejemplo
<<	Recorrer bits a la izquierda, insertando un bit 0 por la derecha. Los bits sobrantes a la izquierda se pierden.	Si tenemos 101 y recorremos un bit a la izquierda el resultado es 1010
>>	Recorrer bits a la derecha, insertando un bit 0 por la izquierda. Los bits sobrantes a la derecha se pierden.	Si tenemos 101 y recorremos un bit a la derecha el resultado es 10
&	Realiza una operación lógica *and* bit a bit	101&110 = 100
|	Realiza una operación lógica *or* bit a bit	101|110 = 111
^	Realiza una operación lógica *xor* bit a bit	101 ^ 110 = 011

Analicemos uno a uno esto operadores y veamos el uso de cada uno de ellos.

3.2.1. El operador de desplazamiento

El operador de desplazamiento es el que permite recorrer bits a la izquierda o derecha. Si definimos un número entero *int* $i = 10$ la representación binaria del contenido de la variable i es 1010.

El operador de desplazamiento es un operador binario. Esto significa que tiene dos operadores, un número y la cantidad de bits a desplazar. Veamos el código siguiente:

```
int i = 10;
i=i<<1;
```

Esto significa que los bits de la variable i rccorrerán a la izquierda un lugar. El resultado será 20 en decimal o 10100 en binario.

Si utilizamos el operador $i >> 1$ se desplazaran los bits a la derecha dando como resultado 101 cuyo equivalente decimal es 5.

La base de los números binarios es 2, por lo tanto, agregar un cero a la derecha es equivalente a multiplicar por dos. Eliminar un bit de la derecha es equivalente a dividir por 2. Recorrer 2 bits a la izquierda es equivalente a multiplicar por 2 dos veces o sea multiplicar por 2^2, recorrer 3 bits, multiplicar por 2^3, así sucesivamente. Similarmente ocurre lo mismo al recorrer a la derecha, pero dividiendo en lugar de multiplicar.

3.2.2. El operador lógico *and*

El operador lógico *and* se representa con el símbolo & y permite realizar esta operación lógica bit a bit. La tabla siguiente muestra los resultados de realizar una operación *and* entre dos bits:

Operación	resultado
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

Si nos fijamos en el operador & vemos que solo cuando ambos bits son 1 el resultado es 1. Esto es equivalente a multiplicar ambos bits. ¿Cuál es el uso para este operador? Este operador se utiliza para averiguar el valor de uno o más bits. Por ejemplo si tenemos una secuencia de bits y realizamos una operación *and*

```
xxxxyxyyxxyx
000011110000
------------ and
0000yxyy0000
```

No conocemos que son los bits x y y pero sabemos que el resultado será el mismo bit si hacemos & con 1 y 0 si hacemos la operación & con 0. La secuencia de bits 000011110000 del ejemplo se denomina máscara.

Si queremos averiguar si un número es par, podemos simplemente preguntar si el último bit es cero y esto se puede hacer con una operación *and* donde todos los bits se colocan en cero y el bit de más a la derecha en 1. Si el resultado es cero el número será par, si es uno impar.

3.2.3. El operador lógico *or*

El operador lógico *or* se representa por | y realiza la operación que se muestra en la tabla bit a bit:

Operación	resultado
0|0	0
0|1	1
1|0	1
1|1	1

Esta operación permite colocar un bit en 1 vea que cada vez que se hace | con 1 el resultado es siempre 1. Cuando queremos colocar bits en uno usamos esta instrucción.

3.2.4. El operador lógico *xor*

El operador lógico *xor* se representa por ^ y realiza la operación *xor* que se muestra en la tabla bit a bit:

Operación	resultado
0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

Esta operación es equivalente a sumar los bits. Si realizamos una operación *xor* con de una variable consigo misma es equivalente a poner la variable en cero. Una de las aplicaciones del *xor* es el cambiar todos los bits de una variable, lo unos por ceros y los ceros por unos. Si realizamos una operación *xor* con todos los bits en 1 se obtendrá esto. Por ejemplo 1010 ^ 1111 dará el resultado 0101.

Otra característica es que si realizamos *xor* con un valor dos veces se obtiene otra vez el valor original. Veamos, por ejemplo, 43 = 101011, 57 = 111001 si hacemos 43 ^ 57 el resultado es 18 = 010010, ahora si hacemos 43 ^ 18 obtenemos otra vez 57. Del mismo modo 57 ^ 18 = 43.

3.2.5. Aplicación de manejo de bits

Una de las aplicaciones del manejo de bits es el de mostrar un número entero por pantalla en binario. Tomemos por ejemplo el número 43 = 101011. Vemos el código

```
int i = 43;
System.out.print(i);
```

Por pantalla se verá el número 43. Ahora si queremos mostrar el equivalente en binario, tenemos que trabajar bit a bit, y realizamos lo siguiente:

1. Creamos una máscara con un 1 en el primer bit que debemos mostrar
2. Realizamos una operación *and*
3. Recorremos este bit al extremo derecho
4. Mostramos el bit
5. Recorremos la máscara un lugar a la derecha

6. Repetimos el proceso con todos los bits

Para el ejemplo el código seria:

```
int i = 43;
System.out.print(i +"=");
// mostramos el bit 5
int mascara = 32;
int bit = i & mascara;
bit = bit >> 5;
System.out.print(bit);
// mostramos el bit 4
mascara= mascara >>1;
bit = i & mascara;
bit = bit >> 4;
System.out.print(bit);
// mostramos el bit 3
mascara= mascara >>1;
bit = i & mascara;
bit = bit >> 3;
System.out.print(bit);
// mostramos el bit 2
mascara= mascara >>1;
bit = i & mascara;
bit = bit >> 2;
System.out.print(bit);
// mostramos el bit 1
mascara= mascara >>1;
bit = i & mascara;
bit = bit >> 1;
System.out.print(bit);
// mostramos el bit 0
mascara= mascara >>1;
bit = i & mascara;
System.out.print(bit);
```

Para mostrar una variable entera en binario podemos usar la clase *Integer* y el método *toBinaryString* con lo cual *Integer.toBinaryString*(43) = 101011, pero como dijimos, muchas veces es más eficiente resolver un problema utilizando operaciones en binario.

3.3. Trabajando con variables

Para trabajar directamente en números sin preocuparnos de que internamente están definidos en binario, tenemos las siguientes operaciones binarias:

Operación	Descripción
+	Suma de dos variable
-	Resta de dos variables
/	División de dos variables
%	Hallar del resto de una división

y las siguientes operaciones incrementales

Operación	Descripción
++	Incrementa en uno
- -	Decrementa en uno

Para realizar operaciones aritméticas entre dos variables del mismo tipo simplemente utilizamos el operador deseado y asignamos el resultado a otra variable. Por ejemplo para sumas A con B y guardar el resultado en C escribimos $c = A + B$.

La expresión 2+3*5 puede dar dos resultados de acuerdo al orden en que se realicen las operaciones. Si primero hacemos la suma $2 + 3 = 5$ y multiplicamos por 5 el resultado es 25 si primero hacemos la multiplicación $3 * 5 = 15$ y luego la sumamos 2 el resultado es 17. El resultado debe evaluarse en función de la prioridad de los operadores. Primero se evalúa la multiplicación y después la suma.

Los operadores que realizan operaciones incrementales, solo trabajan sobre una variable. Por ejemplo $A++$ incrementa la variable A en uno.

Los operadores se evalúan de acuerdo al orden de precedencia que se muestra en la siguiente tabla. Si tienen el mismo orden de precedencia se evalúan de izquierda a derecha.

Prioridad	Operador
1	( )
2	++ -
3	* / %
4	+ -
5	>>, <<
8	&
6	^
7	|

3.3.1. Ejemplos de expresiones

Las siguientes expresiones se evalúan de acuerdo a la prioridad de los operadores y dan los resultados mostrados:

Expresión	Resultado
$3 + 2 * 3$	Primero se realiza la multiplicación luego la suma el resultado es 9
$(3 + 2) * 3$	Primero se realizan las operaciones entre paréntesis luego la multiplicación el resultado es 15
$(3 + 2 * 3) * 2$	Primero se realiza lo que esta entre paréntesis de acuerdo a la prioridad de operaciones, dando 9 luego se multiplica por 2 el resultado es 18
$3/2 * 3$	Como las operaciones son de la misma prioridad primero se hace 3/2 como son números enteros el resultado es 1 luego se multiplica por 3 dando 3
$3 * 3/2$	Primero se hace $3 * 3 = 9$ dividido por 2 da 4
3,0/2,0 * 3,0	Como los números son de punto flotante 3,0/2,0 el resultado es 1,5 por 3 da 4.5
$++a + b$	Primero se incrementa el valor de a y luego se suma b
$a + b++$	Primero se suma $a + b$ luego se incrementa el valor de b

3.3.2. La clase Math

Como se ve no hay operadores para hacer raíz cuadrada, exponencial, funciones trigonométricas, etc. Para estas funciones disponemos de la clase *Math*. La tabla muestra algunos métodos de la clase

Math.

Método	Descripción
Math.abs(a)	Devuelve el valor absoluto de a
Math.sqrt(a)	Devuelve la raíz cuadrada de a
Math.pow (a,b)	Devuelve el valor de a^b
Math.PI	Devuelve la constante PI
Math.hypot(a, b)	Devuelve $\sqrt{a^2+b^2}$
Math.max(a,b)	Devuelve el máximo entre a y b
Math.min(a,b)	Devuelve el mínimo entre a y b

Por ejemplo, dados dos puntos A, B dados por sus coordenadas a_1, a_2 y b_1, b_2,para hallar la distancia entre dos puntos estos puntos escribimos la expresión: $Math.hypot((a_1 - b_1), (a_2 - b_2))$.

3.4. Operadores de asignación

Para facilitar la escritura de expresiones se han creado los operadores de asignación. Por ejemplo si necesitamos realizar una operación con una variable y deseamos guardar el resultado en la misma variable utilizamos los operadores de asignación. Por ejemplo $a+ = b$ es equivalente a $a = a + b$.

Ejemplo	Equivalencia
$a+ = expresion$	$a = a + expresion$
$a- = expresion$	$a = a - expresion$
$a* = expresion$	$a = a * expresion$
$a/ = expresion$	$a = a/expresion$
$a\% = expresion$	$a = a\%expresion$

3.5. Convertir el tipo de datos

Para realizar conversiones de datos es necesario hacer una *cast*. Veamos algunos ejemplos.

- Para eliminar los decimales de una variable de punto flotante la llevamos a una de tipo entero como sigue

```
float f = 34.56;
int i = (int)f;
```

- Si deseamos que una variable *int* se convierta en *long*

```
int i = 34;
long l = (long)i;
```

- Si hacemos el proceso inverso vale decir convertir de *long* a *int*, solo se obtiene el resultado correcto si el valor puede ser almacenado en una variable entera. El código siguiente claramente da un resultado equivocado.

```
long l = Long.MAX_VALUE/100;
int i = (int)l;
System.out.println(l);
System.out.println(i);
```

El valor de l es 92233720368547758 y el resultado en la variable i es 2061584302. Claramente incorrecto.

- En los números de punto-flotante se trabaja de la misma forma

```
double d = 55.34;
float f = (float)d;
```

Si el valor no puede almacenarse en la variable f el resultado es *Infinity*.

3.6. Lectura del teclado

Para leer datos del teclado utilizamos la clase *Scanner*. Para poder utilizar esta clase es necesario indicar al compilador donde se encuentra. Esto se realiza con la instrucción *import java.util.Scanner* que se incluye al principio del programa. También se puede colocar *import java.util.** Esto último indica que queremos importar todas las clases de *import java.util*. La conveniencia de especificar una por una las clases es mejorar el tiempo de compilación en programas grandes.

Seguidamente es necesario crear una instancia de la clase *Scanner* para poder utilizar los métodos de la clase y leer del teclado. Esto podemos hacer con *Scanner lee =new Scanner(System.in)*. El nombre *System.in* indica que los datos se ingresaran del teclado.

Para leer una variable entera el código queda como sigue:

```
import java.util.Scanner;
public class programa2 {
    public static void main(String[] args) {
        Scanner lee =new Scanner(System.in);
        int i = lee.nextInt();

    }

}
```

Cuando creamos una instancia de la clase *Scanner* lo que hacemos es iniciar un flujo de datos nuevo desde el teclado. Cada lectura de una variable entera inicia después de los espacios y termina cuando se encuentra el primer espacio, que se utiliza como delimitador entre número. Por ejemplo para leer dos números i, j que están en una línea

```
123   987
```

Codificamos:

```
import java.util.Scanner;
public class programa2 {
    public static void main(String[] args) {
        Scanner lee =new Scanner(System.in);
        int i = lee.nextInt();
        int j = lee.nextInt();

    }

}
```

Si los datos de la entrada están definidos en múltiples líneas el resultado es el mismo.

En caso de que la entrada no iguale con la del tipo de datos que queremos leer el Java dará el error *java.util.InputMismatchException.*

Para leer datos del teclado existen los siguientes métodos:

Método	Descripción
next()	Leer una cadena de caracteres hasta encontrar un espacio
nextInt()	Leer un número entero
nextFloat()	Leer un número de punto flotante
nextDouble()	Leer un número de tipo *Double.*
nextLong()	Leer un número de tipo *Long*
nextLong(base)	Leer un número de tipo *Long* escrito en la *base* especificado.
nextShort()	Leer un número de tipo *Short*
nextLine()	Leer toda una línea y almacenar en una cadena.

Por ejemplo si queremos convertir un número de escrito en base octal, a decimal, podemos utilizar el método *nextLong(3)* luego al imprimir obtendremos el equivalente en decimal. Si ingresamos 120120 al ejecutar el siguiente programna obtendremos 420 que es su equivalente en decimal.

```
import java.util.Scanner;
public class programa2 {
    public static void main(String[] args) {
        Scanner lee =new Scanner(System.in);
        Long i = lee.nextLong(3);
        System.out.println(i);

    }

}
```

Como ve conocer y utilizar los diferentes métodos disponibles simplifica el desarrollo de programas.

3.7. Errores de redondeo

Tipo	Descripción	Tamaño
int	Tipo entero con un rango desde $-2,147,483,648$ hasta $2,147,483,647$	4 bytes
byte	Describe un solo byte con un rango desde -128 hasta 127	1 byte
short	Entero pequeño con un rango desde ?32768 hasta 32767	2 bytes
long	Entero grande con un rango desde $-9,223,372,036,854,775,808$ hasta $9,223,372,036,854,775,807$	8 bytes
double	Doble precisión de punto flotante con un rango de $\pm 10^{308}$ precisión de 15 dígitos decimales	8 bytes
float	Simple precisión de punto flotante con un rango de $\pm 10^{38}$	
char	Tipo carácter representando la codificación ascii: definida en el equipo	2 bytes
boolean	Valor que admite solo verdadero o falso	1 bit

En muchos casos se producen errores de redondeo cuando no se pueden convertir exactamente los datos.

```
double f = 4.35;
System.out.println(100 * f);
// Imprime 434.99999999999994
```

Utilice el método round para redondear números de punto flotante.

```
long redondeado = Math.round(balance);
```

Para convertir de tipo utilice cast colocando el tipo entre paréntesis.

```
(int) (balance * 100)
```

3.8. Ejercicios

Para todos los ejercicios siguientes debe ingresar los datos por teclado y mostrar los resultados por pantalla en una sola fila.

1. Escriba un programa que dado un carácter en letras mayúsculas y luego mostrar por pantalla el carácter en letras minúsculas. Para esto utilice una instrucción *Xor*, que le permitirá cambiar de mayúsculas a minúsculas y de minúsculas a mayúsculas con la misma intención. Para leer un solo carácter del teclado se utiliza la instrucción *System.in.read()* cuyo resultado es un número entero.

   ```
   Ejemplo de entrada
   A
   Ejemplo de salida
   a
   ```

2. Escriba un programa que lea un carácter del teclado e imprima 0 cuando la letra es mayúscula y 1 cuando es minúscula. para esto utilice la instrucción *and* y desplazamiento de bits.

```
Ejemplo de entrada
A
Ejemplo de salida
0
```

3. Escriba un programa que lea un número del teclado y con manejo de bits imprima uno o cero según sea par o impar imprima 0, 1 respectivamente.

```
Ejemplo de entrada
123
Ejemplo de salida
1
```

4. Ingrese un número por teclado luego desplace 3 bits a la derecha. Muestre el número. Luego escriba una instrucción de división que logre el mismo resultado.

5. Escriba un programa java que dados dos números enteros imprima

   - La diferencia
   - El producto
   - El promedio
   - La distancia, el valor absoluto de la diferencia
   - El máximo
   - El mínimo

```
Ejemplo de entrada
5 7
Ejemplo de salida
-2 35 6.0 2 7 5
```

6. Escriba un programa que dado el radio de una circunferencia halla la superficie. La ecuación de la para hallar la superficie es $pi * radio^2$.

```
Ejemplo de entrada
3
Ejemplo de salida
28.274333882308138
```

7. Escriba un programa que dada la altura y el diámetro en centímetros de una lata cilíndrica de refresco calcule el volumen.

```
Ejemplo de entrada
10 7
Ejemplo de salida
384.84510006474966
```

8. Dados dos puntos $A(x_1, y_1), B(x_2, y_2)$ en el plano la distancia entres A, B se define como $d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Dados los puntos A, B mostrar la distancia entre ambos.

```
Ejemplo de entrada
-7 4 6 2
Ejemplo de salida
13.152946437965905
```

9. Escriba un programa que dados dos puntos $A(x_1, y_1), B(x_2, y_2)$ escriba la ecuación de la recta que pasa por ellos.

```
Ejemplo de entrada
0 1 5 6
Ejemplo de salida
y = x + 1
```

10. Escriba un programa que dados dos puntos (centro) $A(x_1, y_1), B(x_2, y_2)$ escriba la pendiente de la ecuación de la recta que pasa por ellos. La ecuación es $m = (x_1 - x_2)/(y_1 - y_2)$

```
Ejemplo de entrada
6 5 3 2
Ejemplo de salida
13.152946437965905
```

11. Dadas dos circunferencias representadas por un punto y su radio que no se intersectan. Escriba un programa que despliegue el centro y radio de la circunferencia más pequeña que interseeta a ambas.

```
Ejemplo de entrada
0 1 2
0 6 1
Ejemplo de salida
0 4 1
```

12. La figura 3.1 muestra un triángulo rectángulo que se utilizará en este y los siguientes ejercicios. Ingrese los valores de a y β halle el valor de c

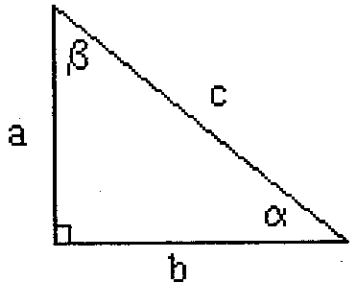


Figura 3.1: Triángulo rectángulo para los ejercicios.

13. Ingrese los valores de a, b, c y muestre el perímetro del triángulo.

14. Ingrese los valores de b y α y calcule el valor de c.

Capítulo 4

Estructuras de control

4.1. Introducción

Los lenguajes de programación implementan tres tipos de instrucciones

- Instrucciones de asignación
- Instrucciones de iteración
- Instrucciones de comparación

Los operadores de asignación ya fueron explicados, permiten asignar un valor a una variable. Los operadores de iteración permiten repetir bloques de instrucciones. Y los operadores de condición se utilizan para comparar valores. En el capítulo explicaremos éstas instrucciones y el uso en la resolución de problemas.

4.2. Agrupamiento de instrucciones

Las instrucciones se pueden agrupar en bloques con el uso de corchetes {}. Esto permite que las definiciones de variables tengan un ámbito de vigencia. Vemos el código siguiente:

```
{
int i=1;\textbf{}
System.out.println(i);
}
System.out.println(i);
```

La variable i está definida dentro de un bloque y por esto solo es accesible en su interior. Esto se puede apreciar en las instrucciones *System.out.println(i)*. La primera instrucción permite mostrar el valor de i en pantalla. La segunda que esta fuera del bloque muestra el error *cannot be revolved* que indica que es una variable no accesible en este lugar.

Si queremos hacer esta variable disponible dentro y fuera del bloque hay que definirla fuera del bloque.

```
int i=1;
{
System.out.println(i);
}
System.out.println(i);
```

Las variables que se definen dentro del bloque se denominan variables locales. En otros casos se denominan variables globales.

4.3. Estructuras de control condicionales

Las estructuras de control condicionales son las que nos permiten comparar variables para controlar el orden de ejecución de un programa. Estas estructuras de control implementan con la instrucción *if*, *if else*, *if else if*, y *?*. Estas instrucciones nos permiten decidir que ciertas instrucciones que cumplan una determinada condición se ejecuten o no.

4.3.1. Estructura de control *if*

El diagrama 4.1 muestra el flujo de ejecución de una instrucción *if*.

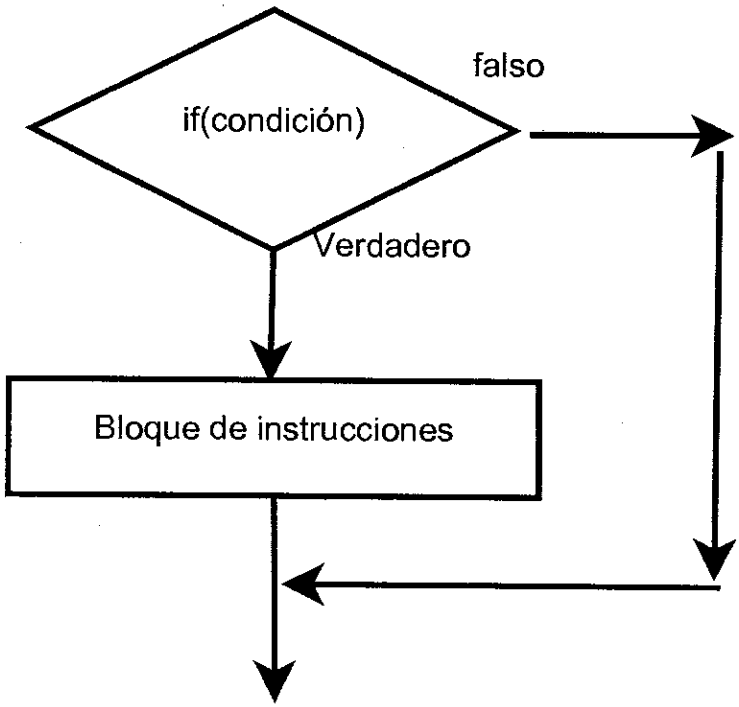


Figura 4.1: Flujo de ejecución de un *if*

La sintaxis de Java es:

```
if (condición) {
   bloque de instrucciones
   }
```

o

```
if (condición)
   instrucción;
```

Note que después de los paréntesis del *if* no se coloca un punto y coma.

4.3.2. Operadores condicionales

Los operadores condicionales para comparar variables se muestran en la tabla siguiente:

Operador	Descripción
==	igual
<	menor que
<=	menor o igual
>	mayor que
>=	mayor o igual
!=	no igual
!	negar

Los operadores condicionales trabajan con dos variables y el resultado puede ser verdadero (true) o falso (false). Por ejemplo $a > b$, $a == b$, etc. Hay que tomar en cuenta que los operadores ==, <=, > 0, != se escriben sin espacios.

Ejemplo.

Supongamos que queremos resolver la ecuación $ax + b = 0$. Se ingresan los valores de a, b por teclado, y se imprime el valor calculado de x. La solución de esta ecuación es $x = -b/a$. Como sabemos solo existe si a es mayor que cero. Para resolver esto realizamos el siguiente programa:

```
import java.util.Scanner;
public class Ecua {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int a= lee.nextInt();
        int b= lee.nextInt();
        if (a !=0)
            System.out.println((float)-b/a);
    }
}
```

La instrucción *if (a!=0)* verifica que se realizará una división por cero. Cuando a es diferente a cero se calcula el resultado de x. Cuando $a = 0$ no obtiene ningún resultado.

4.3.3. Estructura de control *if else*

El diagrama 4.2 muestra el flujo de ejecución de una instrucción *if else*.

La sintaxis de Java es:

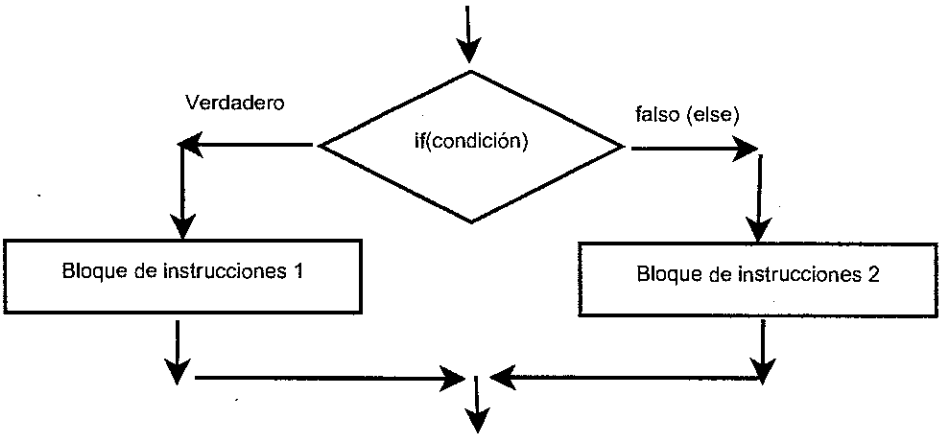


Figura 4.2: Flujo de ejecución de un *if else*

```
if (condición) {
   bloque de instrucciones 1
   }
else {
bloque de instrucciones 2
}
```

o

```
if (condición)
   instrucción1;
   else
   instrucción2;
```

Esta instrucción nos dice que si cumple la condición se ejecuta el bloque de *instrucciones 1* y si no se cumple se ejecuta la secuencia de *instrucciones 2*.

Consideremos el ejemplo anterior para hallar la solución de $ax + b = 0$. Utilizando la estructura *if else* podemos mostrar un mensaje indicando que no existe solución. El código es:

```
import java.util.Scanner;
public class Ecua {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int a= lee.nextInt();
        int b= lee.nextInt();
        if (a !=0)
            System.out.println((float)-b/a);
         else
            System.out.println(''No existe una solucion");
    }
}
```

4.3.4. Estructura de control *if else if*

Muchas veces debemos anidar las instrucciones *if*. Esto se puede hacer como sigue:

```
if (condicion1){
    instrucciones1
    if (condicio2){
        instrucciones2
    }
    else {
        instrucciones2
    }
    else {
        instrucciones3
    }
```

Este forma de anidar se puede repetir las veces que sea necesario sin un límite. Ahora podemos mejorar el programa para resolver $ax + b = 0$ considerando el caso en que $a = b = 0$ en el cual pondremos un mensaje apropiado. El programa queda como sigue:

```
import java.util.Scanner;
public class Ecua {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int a= lee.nextInt();
        int b= lee.nextInt();
        if (a !=0)
            System.out.println((float)-b/a);
        else {
            if (b==0)
                System.out.println(''No puede ingresar a y b en 0");
            else
                System.out.println(''No existe una solucion");
        }
    }
}
```

4.3.5. Conectores lógicos *and, or*

Las expresiones lógicas pueden asociarse a través de conectores lógicos *and* y *or*. En java éstos conectores representan por && y || respectivamente. Vea que son diferentes a las operaciones con bits que tienen un solo símbolo.

Con estas instrucciones podemos realizar comparaciones más complejas. Por ejemplo $(a > b)\&\&(c > d)$. Que significa que debe cumplirse la primera condición y la segunda condición simultáneamente.

La tabla siguiente muestra como trabaja el operador *and*.

Expre.A	Operador	Expre.B	Resultado
true	&&	true	true
true	&&	false	false
false	&&	true	false
false	&&	false	false

La tabla siguiente muestra como trabaja el operador *or*.

Expre.A	Operador	Expre.B	Resultado
true	||	true	true
true	||	false	true
false	||	true	true
false	||	false	false

La tabla siguiente muestra como trabaja el operador *not*.

Operador	Expre.	Resultado
!	true	false
!	false	true

4.3.6. Prioridad de los operadores

Los conectores lógicos se procesan de acuerdo al siguiente orden de prioridades: Primero los paréntesis las instrucciones *and* y después las *or*. Veamos un ejemplo: Sea $a = 1, b = 2, c = 1$ y deseamos evaluar la expresión $(a > b)\&\&(b < c) \,||\, (a == c)$

$a > b$	Oper.	$b < c$	Oper.	$a == c$
false	&&	false	||	true

Primero realizamos las operaciones *and* y tenemos $false\&\&false$ cuyo resultado es $false$. Continuamos con la operación *or* y tenemos que $false \,||\, true$ que da *true*. Para los valores datos esta expresión dará como resultado verdadero.

Si queremos cambiar el orden de evaluación debemos usar los paréntesis quedando $(a > b)\&\&((b < c) \,||\, (a == c))$. En este caso operación *or* da $false \,||\, true$ es *true* y luego haciendo *and* con $false$ el resultado es $false$, que como ve, es diferente al anterior.

4.3.7. Propiedades y equivalencias

En la utilización de los operadores lógicos es necesario conocer las siguientes equivalencias.

Expresión	Equivalencia	Descripción
$!(!a)$	a	Doble negación
$!(a\&\&b)$	$!a \,||\, !b$	negación de un *and*
$!(a \,||\, b)$	$!a\&\&!b)$	negación de un *or*

Las equivalencias dos y tres, se denominan leyes de De Morgan. Cuando tenemos una serie de condiciones podemos aplicar estas equivalencias para simplificar las expresiones lógicas.

4.3.8. Estructura de control ?

La estructura de control ? es equivalente a una estructura *if else*. La sintaxis es:

varibale=condicion ? expresión por verdad : expresión por falso;

Por ejemplo si deseamos hallar el máximo entre a y b y guardar el resultado en la variable m, utilizando *if else* la codificación es como sigue:

```
if (a>b)
    m=a;
else
    m=b;
```

Utilizando la estructura *?* la sintaxis para hallar el máximo es:

```
m=a>b?a:b;
```

Lo que expresa esta instrucción es que si $a > b$ entonces m toma el valor de a en otros casos toma el valor de b.

4.3.9. Estructura de control *switch*

Para introducir la instrucción *switch* supongamos que tenemos las notas de $0-5$, y queremos imprimir el texto reprobado para las notas $1, 2, 3$, el texto *raspando* cuando la nota es 4, y aprobado cuando es 5. Utilizando una secuencia de instrucciones *if else* se codifica como sigue:

```
import java.util.Scanner;
public class EjeSwitch {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
         int nota=lee.nextInt();
         if (nota==1)
             System.out.println(''Reprobado");
         else
             if (nota==2)
             System.out.println(''Reprobado");
         else
             if (nota==3)
             System.out.println(''Reprobado");
         else
             if (nota==4)
             System.out.println(''Raspando");
         else
             if (nota==5)
             System.out.println(''Aprobado");
    }
}
```

Este código como ve es muy poco entendible, poco modificable, y propenso a error.

La instrucción *switch* permite resolver estos problemas. La sintaxis es

```
switch (variable) {
      case valor entero : instrucciones; break;
      case valor entero : instrucciones; break;
      case valor entero : instrucciones; break;
}
```

Esta instrucción trabaja exclusivamente sobre un valor entero. En cada instrucción *case* se compara el valor indicado con el valor de la variable. Si es menor o igual se ingresa a las instrucciones que están después de los dos puntos. La instrucción *break* hace que la secuencia continúe al final del bloque. El ejemplo anterior utilizando la instrucción *switch* se codifica como sigue:

```
import java.util.Scanner;
public class EjeSwitch {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
         int nota=lee.nextInt();
         switch (nota){
             case 1:
             case 2:
             case 3: System.out.println(``Reprobado"); break;
             case 4: System.out.println(``Raspando"); break;
             case 5: System.out.println(``Aprobado"); break;

             }
    }
}
```

Como se ve es una codificación más simple de entender.

4.4. Estructuras de control iterativas

Las instrucciones de iteración, son instrucciones que permiten repetir varias veces un bloque de instrucciones. Se conocen como estructuras de control repetitivas o iterativos. También se conocen como bucles o ciclos. Nos permiten resolver diversos tipos de problemas de forma simple.

Supongamos que deseamos sumar una cantidad de números leídos del teclado. Es posible escribir un programa que lea un número y sume, luego lea otro número sume, etc. Esta estrategia genera los siguientes problemas:

- El código sería muy largo
- Si aumentamos más datos a sumar debemos aumentar las instrucciones
- Es propenso a tener errores
- Si hay que hacer cambios puede ser muy moroso
- Muchas partes del programa van a estar duplicadas.

4.4.1. Ciclo for

El ciclo for tiene la siguiente sintaxis:

```
for (expresión 1; expresión 2; expresión 3) {
     bloque de instrucciones
     }
```

El proceso de esta instrucción es como sigue:

1. Se ejecutan las instrucciones definidas en expresión 1
2. Se evalúa la expresión 2. Si es verdadera se ejecuta el bloque de instrucciones
3. Se ejecutan las instrucciones de expresión 3 y se vuelve a 2

El diagrama 4.3 muestra un diagrama de como se ejecuta el ciclo *for*. Veamos algunos ejemplos de

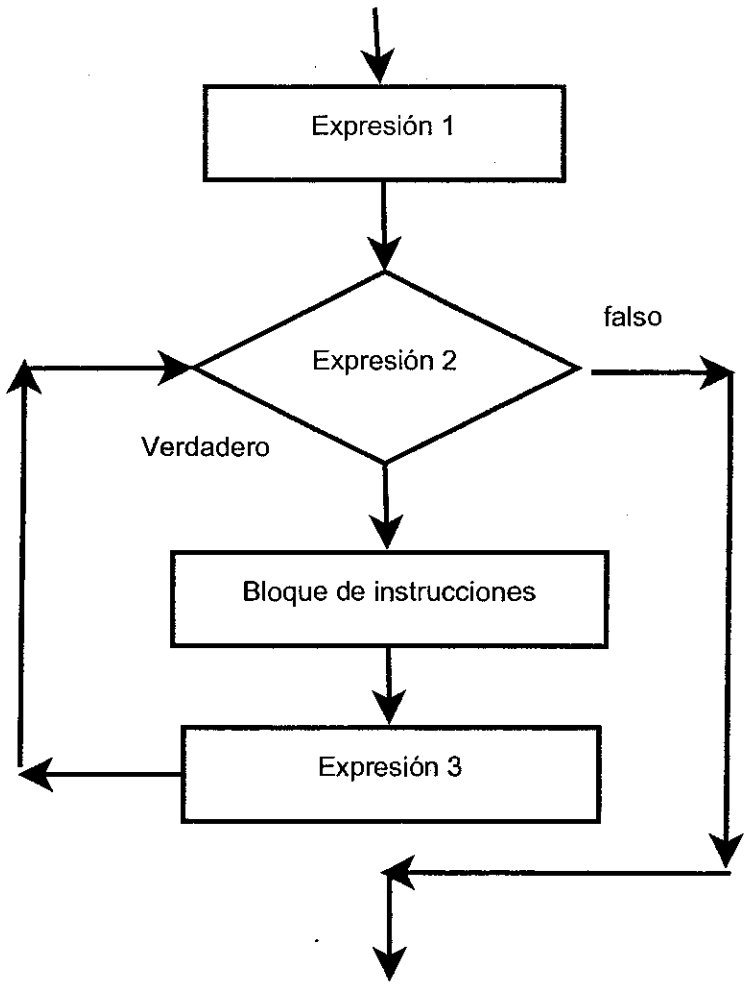


Figura 4.3: Diagrama de la instrucción for

utilización. Queremos calcular el factorial de un número n, que se define como $n! = (n)(n-1)(n-2)...,1$. Para esto hay que construir un bucle con un valor que varía desde 1 hasta n. Dentro del ciclo se realiza la multiplicación.

En la *expresión 1* crearemos una variable que nos servirá para contar el número de iteraciones, que en este caso es n. Esto hacemos con la instrucción $inti = 1$. Al poner *int* como tipo para la variable i hacemos que i sea una variable local dentro del bloque de instrucciones.

Luego definimos la *expresión 2* que regula la finalización del ciclo. En este caso debemos hacer variar el valor de i hasta que tome el valor de n. La condición para controlar la finalización es $i <= n$. Mientras que se cumpla esta condición se ejecuta el bloque de instrucciones.

Finalmente viene la *expresión 3*. Aquí incrementamos el valor de i en uno con la instrucción $i++$.

Ahora el código que resuelve el problema. Iniciamos una variable que utilizaremos para hallar el factorial. Esta variable la denominaremos f y la inicializamos en 1. Luego podemos multiplicar por i y así al finalizar el ciclo *for* f tendrá el factorial de n. El código para calcular el factorial de 5 es:

```
public class fact {
    public static void main(String[] args){
        int n=5,f=1;
        for (int i=1;i<=n;i++){
            f=f*i;
        }
        System.out.println(f);
    }
}
```

Hay que ver que la variable f y la variable n fueron definidas fuera del ciclo. Si nos fijamos luego de la instrucción *for* no se coloca un punto y coma para evitar que la misma termine antes de iniciado el bloque de instrucciones a ejecutar.

Una forma abreviada de escribir puede ser:

```
public class fact {
    public static void main(String[] args){
        int f=1;
        for (int i=1,n=5;i<=n;f=f*i,i++);
            System.out.println(f);
    }
}
```

En este código en el primer bloque de instrucciones hemos agregado la definición de la variable n con su valor inicial 5. En el tercer bloque hemos agregado la instrucción que calcula el factorial. Con esto no se requiere definir un bloque, todo queda contenido en la instrucción *for*. Esta es la razón por la que colocamos un punto y coma al final la instrucción.

Veamos otro ejemplo. Se quiere hallar el resultado de la expresión:

$$s = \sum_{i=0}^{i=10} i^2$$

Para este problema inicializamos una variable s en 0. Luego con un for podemos variar i en todo el rango de la suma. El código resultante es:

```
public class suma {
    public static void main(String[] args){
        int s=0;
        for(int i = 0;i<=10;i++){
            s+=(i*i);
        }
        System.out.println(s);
    }
}
```

Ahora si conocemos una expresión que nos de directamente el resultado de la suma, el programa es más eficiente. Dado que demora mucho menos tiempo de proceso. En muchas expresiones esto es cierto, en cambio para otras no es posible. En el ejemplo que mostramos es posible:

$$s = \sum_{i=0}^{i=n} i^2 = n(n+1)(2n+1)/6$$

4.4.2. Ciclo while

El ciclo *while* permite ejecutar las instrucciones de un bloque hasta que se cumpla cierta condición. la codificación de esta instrucción es como sigue:

```
while (expresión) {
    bloque de instrucciones
    }
```

El diagrama 4.4 muestra como se ejecuta el ciclo *while*. Para asegurar que se llegue a la condición de

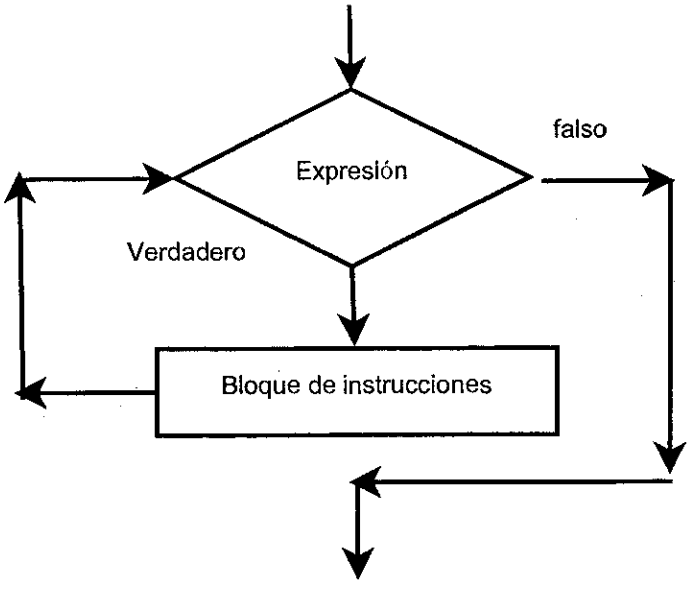


Figura 4.4: Diagrama de la instrucción while

finalización debe existir algo que haga variar la condición dentro del ciclo. El ejemplo que realizamos para hallar el factorial también puede realizarse utilizando una instrucción *while*. el programa siguiente muestra esta implementación.

```
public class fact {
    public static void main(String[] args){
        int n=5,f=1, i=1;
        while (i<=n){
            f=f*i;
            i++;
        }
        System.out.println(f);
```

```
    }
}
```

La relación que existe entre un ciclo *for* y *while* es como sigue:

```
for (expresión 1; expresión 2; expresión 3) {
   bloque de instrucciones;
   }
```

Equivale a

```
expresión 1;
while(expresión 2) {
     bloque de instrucciones;
     expresión 3;
     }
```

4.4.3. Ciclo do while

El ciclo *do while* se utiliza para garantizar que el flujo del programa pase una vez por el bloque de instrucciones. La sintaxis es como sigue:

```
do {
   bloque de instrucciones
   } while (expresión);
```

El diagrama 4.5 muestra como se ejecuta el ciclo *do while*.

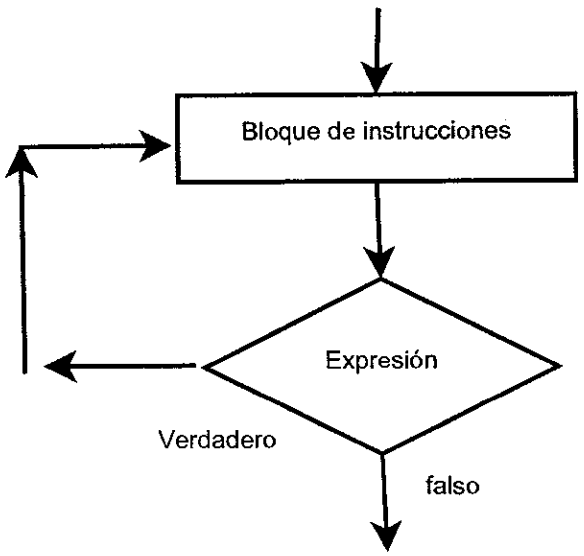


Figura 4.5: Diagrama de la instrucción do while

El programa de factorial mostrado puede también resolverse utilizando un ciclo *do while*. El programa siguiente muestra la forma de resolver el mismo.

```
public class fact {
    public static void main(String[] args){
        int n=5,f=1, i=1;
        do {
            f=f*i;
            i++;
        }
        while (i<=n);
            System.out.println(f);
    }
}
```

4.4.4. Ciclos anidados

Los ciclos pueden anidarse, esto podemos escribir un ciclo dentro de otros ciclos. La forma de anidar bucles se muestra en la figura 4.6.

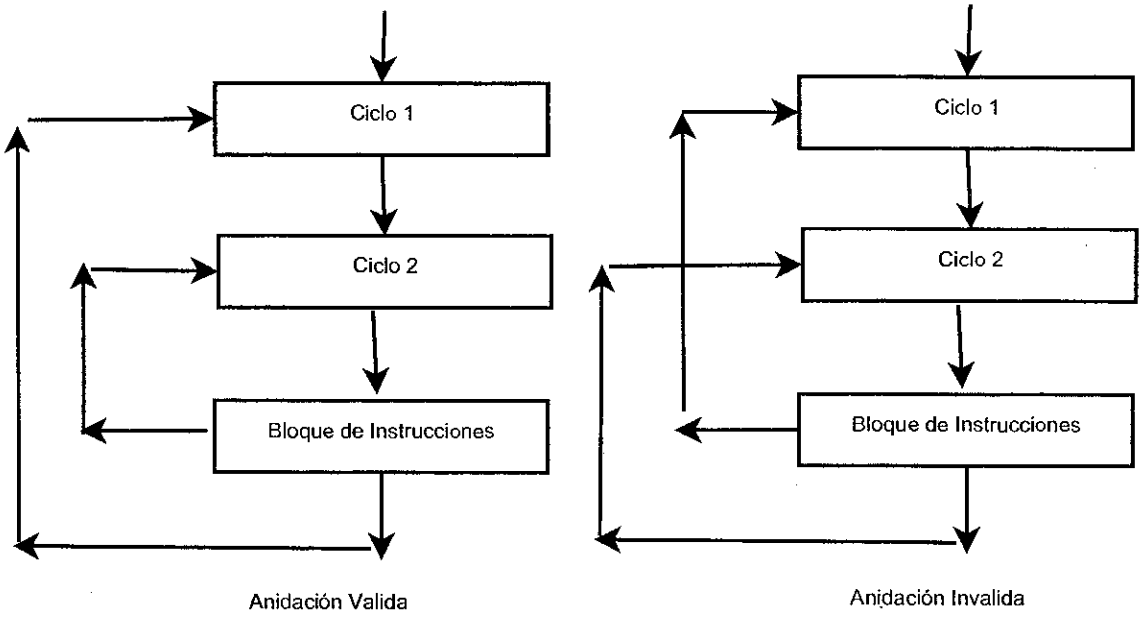


Figura 4.6: Como anidar Ciclos

Veamos un ejemplo de anidar dos ciclos for:

```
public class Anidar {
    public static void main(String[] args){
        for(int i=0;i<5;i++){
            for(int j=0;j<5;j++){
                System.out.print(``("+i+``,"+j+``) ");
            }
            System.out.println();
        }
    }
```

```
}
```

El resultado que produce es:

```
(0,0) (0,1) (0,2) (0,3) (0,4)
(1,0) (1,1) (1,2) (1,3) (1,4)
(2,0) (2,1) (2,2) (2,3) (2,4)
(3,0) (3,1) (3,2) (3,3) (3,4)
(4,0) (4,1) (4,2) (4,3) (4,4)
```

El ciclo exterior hace variar la variable i de 0 *a* 4. El ciclo interior varía $0 \leq j \leq 4$. La instrucción *System.out.print("("+i+","+j+") ")* imprime los valores de i, j sin avanzar a la siguiente línea. Cuando termina el bucle interior avanzamos una línea.

Podemos anidar todo tipo de bucles, respetando siempre que un bucle exterior encierra completamente un ciclo interior. No deben existir bucles cruzados como se muestra en la figura 4.6.

4.5. Lectura de secuencias de datos

Generalmente no se lee un solo valor y debemos leer una secuencia de datos. A esto llamamos proceso de lotes. Supongamos que queremos hallar el promedio de una secuencia de números. Para hacer posible ésto ingresaremos primero la cantidad de datos y luego los datos.

```
5
10 5 80 60 40
```

Los datos vienen separados por espacios para hacer posible que el método *nextInt()* de la clase *Scanner* pueda leer los mismos. Recordamos que el promedio se calcula con la fórmula

$$p = \frac{\sum_{i=1}^{i=n} d_i}{n}$$

Para leer estos datos y calcular el promedio construimos el siguiente programa:

```
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int n = lee.nextInt();
        int s=0, dato;
        for(int i=0;i<n;i++){
            dato=lee.nextInt();
            s=s+dato;
        }
        System.out.println((float)s/n);
    }
}
```

Esto está de acuerdo a la entrada de datos y procesa un solo conjunto de datos. Una posibilidad es que se tengan que ingresar muchas secuencias de las que calcularemos el promedio. Por ejemplo podríamos optar por varias opciones para definir la cantidad de secuencias, Veamos tres posibilidades:

1. Primero especificar la cantidad de casos por ejemplo:

```
2
5
10 5 80 60 40
4
90 20 15 80
```

Hemos anotado en la primera línea la cantidad de secuencias, que son 2, inmediatamente hemos colocado las dos secuencias como en el ejemplo anterior. La solución viene dada por dos ciclos *for* como se muestra en el programa:

```
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int n = lee.nextInt();
        int s=0, dato;
        for(int i=0;i<n;i++){
            dato=lee.nextInt();
            s=s+dato;
        }
        System.out.println((float)s/n);
    }
}
```

2. Una segunda opción es colocar un señuelo al final que indique que hemos llegado al final. Por ejemplo podemos colocar un cero al final para indicar que no hay más secuencias sobre las que queremos trabajar. Para esto colocamos los datos de prueba como sigue:

```
5
10 5 80 60 40
4
90 20 15 80
0
```

Podemos hacer esto de varias formas: Utilizando la instrucción *for*, con la instrucción *while*, y también con *do while*. Expliquemos primero como aplicar un *for* para este caso. Nuestra expresión inicial será leer el número de casos n, luego si éste es mayor a cero, no estamos al final y procesamos el bloque de instrucciones. Finalizado el proceso del bloque de instrucciones, leemos otra vez el número de casos y continuamos. El programa siguiente muestra esta lógica.

```
import java.util.Scanner;

public class programa2 {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        for(int n=lee.nextInt();n>0;n=lee.nextInt()){
            int s=0, dato;
            for(int j=0;j<n;j++){
                dato=lee.nextInt();
                s=s+dato;
            }
            System.out.println((float)s/n);
```

```
        }
    }
}
```

Para resolver esto con la instrucción *while*, primero leemos un valor de n luego, iniciamos un *while* para verificar que llegamos al final. Cuando termina el bloque de instrucciones leemos el próximo valor de n. Esto nos da el código siguiente:

```
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int n=lee.nextInt();
        while(n>0){
            int s=0, dato;
            for(int j=0;j<n;j++){
                dato=lee.nextInt();
                s=s+dato;
            }
            System.out.println((float)s/n);
            n=lee.nextInt();
        }
    }
}
```

Finalmente utilizando la instrucción *do while*. Primero leemos la cantidad de datos del primer caso de prueba. Procesamos y cuando llegamos al *while* leemos otra vez para saber si hay más casos de prueba. El código es el siguiente:

```
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        int n;
        n=lee.nextInt();
        do {
            int s=0, dato;
            for(int j=0;j<n;j++){
                dato=lee.nextInt();
                s=s+dato;
            }
            System.out.println((float)s/n);
        }
        while((n=lee.nextInt())>0);
        }
}
```

3. La tercera posibilidad es que al final esté especificado como *no hay mas datos de prueba*, vale decir, el final del archivo de datos. Los datos estarán dados así

```
5
10 5 80 60 40
4
90 20 15 80
```

Para este caso requerimos de un método de la clase *Scanner*. El método *hasNext* que nos indica que si hay más datos de entrada o no hay más datos. Lo primero que hacemos es colocar un *while* que controle si hay más datos y luego sigue el conjunto de instrucciones. El programa es como sigue:

```
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        while(lee.hasNext()){
            int n=lee.nextInt();
            int s=0, dato;
            for(int j=0;j<n;j++){
                dato=lee.nextInt();
                s=s+dato;
            }
            System.out.println((float)s/n);
        }
    }
}
```

En este método debemos considerar dos casos, primero si leemos del teclado directamente siempre se considera que hay más datos en la entrada. Segundo, cuando es un archivo del disco entonces efectivamente se terminan cuando no hay más datos.

Para que se lean los datos desde un archivo se procesa de la línea de comando como sigue:

```
java Programa < archivo
```

El símbolo < indica que la lectura ya no es del teclado y proviene del archivo. Esto se denomina redireccionar la entrada.

4.6. Ejercicios

1. Polinomios
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Se desea un programa que pueda evaluar polinomios de la forma: $a_1x^n + a_2^{n-1}.....a_n$.

   Por ejemplo si queremos evaluar polinomio $1x^2 + 2x + 3$ con $x = 1$ toma el valor de 6

Input

En la entrada existen varios casos de prueba. La primera línea tiene el valor $0 \leq M \leq 20$ que es el número de elementos del polinomio. Separada por un espacio está el número $0 \leq v \leq 50$ que es donde queremos evaluar el polinomio.

Las siguientes líneas son los coeficientes del polinomio. La entrada finaliza cuando no hay más valores.

Output

Por cada caso de prueba imprima una línea con un número que representa evaluar el polinomio en el punto dado.

Ejemplo de entrada	**Ejemplo de salida**
3 1 1 2 3 4 2 1 0 0 1	6 9

2. Notas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Aún los estudiantes que odian las matemáticas siempre encuentran en éste programa un cálculo muy útil.

¿Cuál es la puntuación más baja que puedo conseguir en la última prueba para sacar un cierta nota? Vamos a escribir un programa para ayudar a reducir al mínimo éste esfuerzo.

Vamos a suponer que, un promedio de 90 o más alto es recompensado con una nota A, 80 o superior (pero menos de 90) es una B, 70 o más (pero menos de 80) es una C, 60 o más (pero menos de 70) es un D. Todos los resultados de las pruebas son enteros entre 0 y 100 inclusive y el promedio no se redondea, por ejemplo, un promedio de 89,99 no consigue una A.

Crear una programa de educación para hallar la nota mínima que se requiere para obtener una nota deseada.

La nota deseada se dará como una cadena de longitud 1, ya sea A, B, C, o D.

Si desea tener una nota de A y tiene 2 notas 0 y 70, aunque obtenga un cien no podrá cumplir su deseo y debe mostrar -1.

Por ejemplo si desea D y tienen 5 notas de 100 inclusive con 0 puede cumplir el mismo.

Input

La Entrada consiste de múltiples casos de prueba, donde cada caso de prueba tiene dos líneas. La primera línea tiene la nota *deseada* y el número de notas que tiene. Las siguiente línea contiene las notas que son un número entero entre 0 y 100.

Output

Se imprimirá en la salida en una sola línea la nota mínima requerida para cumplir su deseo. Si no puede obtener la nota deseada imprima -1.

Ejemplo de entrada	**Ejemplo de salida**
A 2 0 70 D 5 100 100 100 100 100 0 B 4 80 80 80 73 B 5 80 80 80 73 79 80 A 1 80	-1 0 87 80 100

3. Tarifas de Celular
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Usted ha hablado muchos segundos por teléfono y tiene interés de calcular el costo de la llamada. Está establecido que el inicio de una llamada le cuesta 5 centavos, independientemente de la duración de la llamada. Adicionalmente por cada minuto que habla debe pagar. Cualquier fracción de minuto le será cobrada con el precio completo. Así, si usted hablo 65 segundos deberá pagar por 2 minutos. El contador de un minuto nuevo comienza en el segundo 60, 120, 180, etc. termina. Si habla 120 segundos le cobran por 3 minutos.

   El pago por la llamada es un poco complicado. Durante los primeros 5 minutos de la llamada usted debe pagar 10 centavos por minuto. Después de los primeros 5 minutos de la llamada, cada minuto adicional le cuesta 3 centavos.

   Dado la duración de la llamada debe calcular el costo de la misma.

   Ejemplos: Si habla 65 segundos son dos minutos cada uno a 10 hace 20 más inicio de llamada 5 la respuesta es 25.

Input

La entrada consiste en varios casos de prueba. Cada caso de prueba consiste en un número n que representa el número de segundos ($0 \leq n \leq 10000$). La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba imprima una línea con el monto a cobrar.

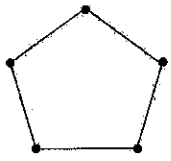
Ejemplo de entrada	**Ejemplo de salida**
65	25
300	58
100	25
0	15
301	58
240	55

4. Crear Pentagonos

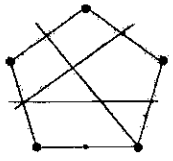
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

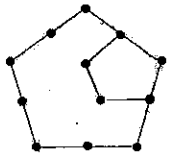
A Pablo le dieron de tarea construir polígonos dentro de otro polígono. El tiene para este propósito muchas canicas pero no sabe cuantas necesitará.

El sabe que para construir un pentágono se requieren 5 canicas.



un pentágono





dos pentágonos

tres pentágonos

La única forma que conoce para insertar un segundo pentágono es colocando una canica en el medio de cada segmento y dibujar tres líneas como se muestra. El pone una canica en cada intersección. Para insertar un tercer pentágono repite el proceso.

Para dibujar un segundo pentágono requiere 12 canicas.Dibujar tres requerirá 22 canicas.

Dada la cantidad de pentágonos a dibujar se requiere conocer la cantidad de canicas requeridas.

Input

La entrada tiene varios casos de prueba. Cada caso de prueba tiene un entero N que indica el número de pentágonos a dibujar ($1 \leq N \leq 10^3$).

La entrada termina en una línea que contiene un 0.

Output

Para cada caso de prueba escriba una línea con la cantidad de canicas requeridas.

Ejemplo de entrada	**Ejemplo de salida**
1 2 3 0	5 12 22

5. Monedas Británicas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Antes del año 1971, Gran Bretaña utilizó un sistema de monedas que se remonta a la época de Carlomagno. Las tres principales unidades de la moneda británica fueron el penique, el chelín, y la libra. Se tenían las siguientes equivalencias, existen 12 peniques en un chelín y 20 chelines en una libra. Dado una cantidad de monedas peniques se quiere convertir esta cantidad en libras, chelines y peniques para esto se procede de la siguiente manera, la primera conversión sera de monedas de peniques a su equivalencia máxima en libras, y luego convertir la mayor cantidad de peniques restantes como sea posible hacia su equivalente en chelines y el restante se mantiene en peniques. Devuelve estos datos en un vector de enteros con tres elementos que contiene la equivalencia en monedas de libras, monedas de chelines, y el número de monedas de un penique, en ese orden.

Input

La entrada de datos consiste en un entero A que representa la cantidad de monedas disponible en peniques, el número está definido como se indica $0 \leq A \leq 10000$. La entrada termina cuando no hay más datos.

Output

Escriba para cada caso de prueba tres números, los cuales son el equivalente en libras, chelín y peniques.

Ejemplo de entrada	**Ejemplo de salida**
533	(2, 4, 5)
0	(0, 0, 0)
6	(0, 0, 6)
4091	(17, 0, 11)
10000	(41, 13, 4)

6. El Juego de Contar

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

El juego de contar se juega entre dos jugadores. El primer jugador comienza a contar desde el 1, y el puede decir a lo mucho un máximo de números. El segundo jugador sigue contando desde donde el primer jugador dejó de contar. Los dos jugadores juegan intercalando los turnos, cada uno aportando al menos un número pero no más que el número máximo(maxAdd). El jugador que dice que el número objetivo es el ganador. Por ejemplo, si maxAdd es 3 y el objetivo es 20, entonces un juego (entre jugadores malos A y B) podrá ser así: $A : 1, 2, 3$, $B : 4, 5$, $A : 6$, $B : 7, 8, 9$, $A : 10, 11, 12$, $B : 13$, $A : 14$, $B : 15, 16$, $A : 17, 18$, $B : 19, 20$. Dado que el jugador B dijo que el número objetivo es el ganador.

Hay una estrategia perfecta para ganar este juego. Cuando maxAdd es 3, si termina su turno diciendo 16, entonces no importa si su oponente contribuya con $1, 2$ *o* 3 números que usted será capaz de contar hasta 20 en su siguiente turno. Del mismo modo, si usted termina su turno diciendo 12, entonces no importa como tu oponente juegue, no podrá terminar su siguiente turno en 16, y así ganará en su siguiente turno.

Crear un método que ayude a jugar en este juego. Dado el maxAdd la mayoría de los números que se puede decir en cada turno, el número objetivo, y el próximo número en el cual debe comenzar a contar. El método debe devolver la cantidad de números que debes decir. Si no hay manera de forzar una victoria, entonces el método debe devolver -1.

Input

La entrada de datos consiste de tres números, **maxAdd** entre $2 \leq maxAdd \leq 8$, el **número objetivo** estará entre $1 \leq num - objetivo \leq 1000$ y el número en el cual debe comenzar a contar, estará entre 1 y el número objetivo.

Output

Para cada caso de entrada mostrar la cantidad de números que debes decir para obtener la victoria y sino se puede lograr el número objetivo mostrar -1.

Ejemplo de entrada	**Ejemplo de salida**
3 20 10 5 20 10 5 20 9	3 5 -1

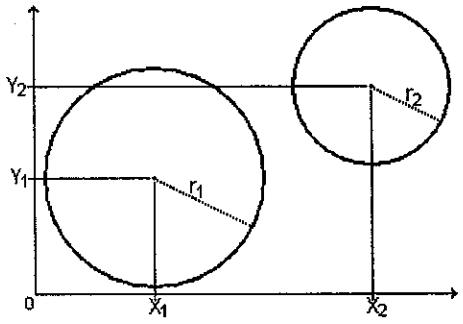
7. Dime Si Intersectan

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Hace una semana, a un curso de niños les dieron un trabajo práctico.

Este trabajo consistía en determinar si un par de circunferencias se intersectan o no.

A cada niño, se le dió los siguientes datos: las coordenadas iniciales (x_1, y_1) y el radio r_1 de la primera circunferencia, y, las coordenadas iniciales (x_2, y_2) y el radio r_2 de la segunda circunferencia.



Cada uno de los niños ha resuelto su trabajo, sin embargo, es el profesor el que no quiere cometer error al momento de calificarlos, por lo que te pide ayuda para saber si los niños han hecho bien su trabajo.

Input

La entrada consiste en un set de datos, que, en cada línea tendrá 6 enteros positivos x_1, y_1, r_1, x_2, y_2 y r_2 separados por un espacio en blanco, con las coordenadas $(0 \leq x_1, y_1, x_2, y_2 \leq 10000)$ y los radios $(0 \leq r_1, r_2 \leq 1000)$

El set de datos termina con una línea con 6 ceros: 0 0 0 0 0 0.

Output

Por cada línea de entrada, usted debe desplegar *SI*, si las dos circunferencias se intersectan y *NO* en caso contrario.

Ejemplo de entrada	Ejemplo de salida
10 10 20 20 20 10 10 10 20 40 40 10 7062 2479 833 6611 3926 744 0 0 0 0 0 0	SI NO SI

8. Fanático del refresco
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Juan es fanático del refresco, pero no tiene suficiente dinero para comprar refrescos. La única forma legal que tiene de adquirir más refresco es juntar las botellas vacías y cambiarlas por más refresco. Adicionalmente a las que consume recolecta botellas en la calle.

Input

La entrada de datos son tres números e, f, c con $e < 1000$ que representa el número de botellas vacías que posee, f representa el número de botellas que halló en la calle $f < 1000$. c representa el número de botellas vacías requeridas para adquirir un refresco. La entrada termina cuando no hay más datos.

Output

Escriba para cada caso de prueba cuantos refrescos pudo tomar en ese día. Cada número debe imprimirse en una línea.

Ejemplo de entrada	**Ejemplo de salida**
9 0 3 5 5 2	4 7

9. Puertas Vaivén

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Todos hemos visto las puertas con vaivén antes, tal vez en la entrada de una cocina. Echa un vistazo a la figura de abajo para dejar las cosas claras. Esta puerta con bisagras en particular, funciona como sigue: La *posición de reposo* es la línea continua , la puerta se inserta en un extremo, y se balancea creando un ángulo (en la imagen, corresponde a la primer swing. Luego, cuando se libera la puerta, se desliza hacia el otro lado (esto se muestra en la imagen como segundo swing). Pero esta vez, el ángulo que oscila se reduce a una fracción conocida del ángulo anterior. Luego, se invierte la dirección de nuevo y, una vez más, el ángulo reducido por la misma fracción. Una vez que el ángulo se reduce a $5,0$ grados o menos, la puerta no oscila más, sino más bien, vuelve a la posición "de descanso".

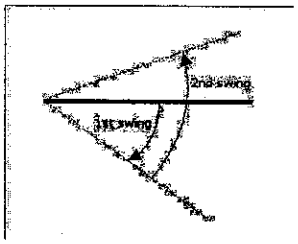


Figura 4.7: Oscilación de la puerta vaivén

Crear una programa que dado un ángulo inicial a desplazarse y una reducción y devuelve un entero correspondiente al número de veces que la puerta se balancea antes de llegar al reposo.

Por ejemplo si se desplaza 50 grados y se reduce en 2 cada vez. Entonces al soltar la puerta el ángulo inicial se ve reducido de $(1/2) * (50) = 25$ grados en el primer vaivén. En este punto, la puerta debe revertir la dirección, y el oscilará a través de un ángulo de $(1/2)*(25) = 12,5$ grados. Continuando de esta manera, la puerta se girará una vez más a través de $(1/2) * (12,5) = 6,25$ grados, y luego a través de $(1/2) * (6,25) = 3,125$ grados. En este punto, la puerta se va a la posición de reposo. Por lo tanto, la respuesta correcta es de 4, ya que la puerta tomó 4 cambios antes de llegar al descanso.

Input

La entrada consiste de varios casos de prueba cada uno en una línea. Cada caso de prueba consiste de dos números enteros separados por un espacio. El primero corresponde al desplazamiento d de la puerta $0 \leq d \leq 90$. Y el segundo a la reducción x en cada oscilación $0 \leq x \leq 10$. La entrada termina cuando no hay más datos.

Output

La salida esta dada por un número entero en una línea que representa el número de oscilaciones que hará la puerta.

Ejemplo de entrada	Ejemplo de salida
50 2	4
45 6	2
23 3	2
3 3	0

10. Adivinanzas

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un juego de adivinanzas muy popular es *suponga el número*, donde una persona selecciona un número en un rango conocido, y otra persona intenta descubrir que número es. Después de cada intento, la primera persona revela si la suposición era correcta, demasiado alta, o demasiado baja.

Muy pronto uno aprende que la mejor estrategia es suponer el número medio entre los cuales aún no han sido descartados. Si no existe un único número medio, entonces existen dos números para seleccionar. En ese caso, nosotros seleccionamos el más pequeño de esos números.

El algoritmo se puede describir así:

```
extremo inferior y limite superior
Repetir
   x = ( el extremo inferior + el límite superior )/2
       (redondee hacia abajo si es necesario)
  Haga x su suposición
  Si  x es demasiado baja, establezca extremo inferior a x+1
  Si  x es demasiado alta, establezca límite superior a x-1
Hasta que x sea correcto
```

Por ejemplo, suponga que los límites inferior y superior son 1 y 9 , respectivamente. El valor medio es 5. Si esto es *demasiado bajo*, los nuevos límites se convierten en 6 y 9. Este rango contiene cuatro números, y como no existe ningún número medio único. De los dos números en el centro que son 7 y 8 escogemos el más pequeño de éstos que es 7 , así nuestra suposición próxima entonces se convierte en 7.

Construya un programa que permita leer el límite superior del rango (el extremo inferior siempre es 1), y el número seleccionado por la primera persona. El programa deberá devolver un entero que representa el número total de intentos requerido por la segunda persona para adivinar el número correcto.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba consiste de dos números el número superior $1 \leq s \leq 1000$ y el número escogido $1 \leq e \leq s$. La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba escriba en la salida un solo número que indica el mínimo numero intentos para adivinar el número escogido.

Ejemplo de entrada	Ejemplo de salida
9 6 1000 750 643 327 157 157 128 64	3 2 7 8 1

11. El pez Dinky

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

El Dinky es un pequeño pez que se cría regularmente, donde cada macho solo tiene una hembra. A los pocos días de su nacimiento, el macho de la especie busca a una compañera y se queda con ella para toda la vida, nunca pierde el tiempo con otra.

Al final de cada mes, las reproducciones de pareja son exactamente dos hijos, de los cuales uno es un macho y otra hembra. Cada uno de estos, a su vez, se va en su búsqueda amorosa. La consanguinidad no es infrecuente en los confines de un tanque de peces o pecera, un par de primos, o incluso hermanos pueden terminar en un apareamiento. Si hay más mujeres que hombres, el número en exceso que no consigue una pareja, no dará a luz en ese mes.

A pesar de sus diminutas dimensiones y naturaleza pacífica, en una población de Dinkies no se debe permitir que se multipliquen hasta el infinito. Los expertos recomiendan asignar por lo menos medio litro de agua por pez diminuto. El tanque se dice que está lleno de peces cuando el espacio es menor que eso. La solución es o bien comprar un tanque más grande o pescar algunos Dinkies para el desayuno.

Dado el volumen de un tanque en litros, el número de Dinkies macho que habitan actualmente en el depósito, y el número de hembras presentes, tiene que calcular el número de meses que puede transcurrir antes de que el tanque se llene de peces. Tenga en cuenta que todas las parejas reproducen simultáneamente al final de cada mes.

Si los valores de entrada son que el tanque ya está lleno de peces, la respuesta correcta es 0. Si el tanque se llena al final del primer mes, la respuesta es 1.

Suponga que todos los Dinkies, jóvenes y ancianos, viven perpetuamente.

Consideremos el ejemplo, $volumen = 10, machos = 4, hembras = 6$, se forman inicialmente cuatro parejas. Al final del primer mes, nacen cuatro machos y cuatro hembras Dinkies. Con lo que hay 18 peces Dinky en total y ocho parejas. Al final del segundo mes, las ocho parejas tienen un par de Dinkies y suman un total de 34, lo que no abastece el medio litro para cada pez en un tanque de diez litros. Así, se ha tomado, dos meses para llegar a un estado lleno de peces.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba está dado en una línea y contiene el volumen del tanque $1 \leq volumen \leq 1000000$, número de peces machos $1 \leq machos \leq 1000$, número de peces hembras $1 \leq hembras \leq 1000$, separados por un espacio. La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada escriba en la salida una línea con un número que indica en cuantos meses se llena el estanque.

Ejemplo de entrada	**Ejemplo de salida**
10 4 6	2
100 4 6	5
5 6 4	1
4 6 4	0
1000000 3 2	19
431131 764 249	11

12. Patrones Mágicos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Ricardo cree que existen patrones mágicos relacionados a algunos números. Esta su creencia deriva de su obsesión de buscar patrones en todo. Recientemente ha descubierto que algunos números tienen una fuente mágica, por ejemplo:

```
     1234
+   12340
+  123400
+ 1234000
---------
  1370974
```

Formalmente 1234 es la fuente mágica de 1370974 porque $1370974 = 1234(1 + \sum_{i=1}^{i=n} 10^i)$.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba consiste de dos números $1 \leq a, b \leq 10000000$ separados por un espacio. La entrada termina cuando no hay más datos.

Output

Para cada caso de prueba imprima en una línea el número más pequeño cuya fuente mágica sea a y que es mayor b.

Ejemplo de entrada	**Ejemplo de salida**
19 200 19 18 333 36963 1234 1000000	209 19 369963 1370974

13. Calcular e

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una fórmula matemática sencilla para calcular el valor de e

$$e = \sum_{i=0}^{n} \frac{1}{n!}$$

En esta ecuación se permite que n vaya al infinito. Esto puede generar aproximaciones precisas de e con valores pequeños de n.

Input

No hay datos de entrada

Output

La salida consiste de los valores aproximados a 9 decimales desde 0 hasta 9 inclusive. La salida debe ser de formato exactamente a la muestra, e imprimirse con la cantidad de decimales especificados

Ejemplo de entrada	**Ejemplo de salida**
	ne - ---------- 0 1 1 2.0 2 2.5 3 2,666666667 4 2,708333333 5 2,716666667 6 2,718055556 7 2,718253968 8 2,718278770 9 2,718281526

14. Densidades

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un farmacéutico está realizando una mezcla de diferentes ingredientes. Necesita conocer la densidad de la mezcla resultante. La densidad se define como la masa dividida entre el volumen.

Cada fila que describe un elemento está en una línea que tiene el *volumen* en *ml*, el *nombre*, la *masa* en gramos. El volumen y la masa son enteros.

Se pide hallar la densidad en gramo por mililitro.

Input

Existen varios casos de prueba. La primera línea de un caso de prueba es un entero que tiene el número de ingredientes $1 \leq i \leq 50$ de la mezcla. La segunda línea tiene la descripción de un elemento. La entrada termina cuando no hay más datos. Cada *volumen* está en el rango $1 \leq v \leq 1000$ y cada *masa* está en el rango $1 \leq m \leq 1000$

Output

Por cada caso de prueba escriba una línea con la densidad, con 10 decimales.

Ejemplo de entrada

```
1
200 ml of oil, weighing 180 g
2
100 ml of dichloromethane, weighing 130 g
100 ml of sea water, weighing 103 g
2
1000 ml of water, weighing 1000 g
500 ml of orange juice concentrate, weighing 566 g
1
1000 ml of something  l i g h t, weighing 1 g
```

Ejemplo de salida

```
0,9000000000
1,1650000000
1,0440000000
0,0010000000
```

15. Potencias

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Se dice que un número y es la kesima potencia de un número x si $y = x^k$

Le dan dos números I, D, Halle el k más grande tal que, $I < y < D$ donde y es la kesima potencia de de algún valor x

Por ejemplo:

Si $I = 5$ y $D = 20$ la potencia más grande entre $5, 20$ es $16 = 2^4$ por lo que la respuesta es 4

Cuando no exista tal valor imprima 1. Si $I = 10$ y $D = 11$ la respuesta es 1.

Input

La entrada consiste de varios datos de prueba. Cada uno tiene dos valore $0 \leq I, D \leq 1000000000$. Termina cuando no hay mas datos

Output

Por cada caso de prueba imprima una línea con el kesimo máximo.

Ejemplo de entrada	**Ejemplo de salida**
5 20 10 12 2 100 1000000000000 1000000000000	4 1 6 12

16. Factoriales

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A usted le dan un número entero N. El factorial de N se define como $N(N-1)(N-2)..,1$. Calcule el factorial de N, quite todos los ceros de la derecha. Si el resultado tiene más de K dígitos, devuelva los últimos K dígitos, en los otros casos devuelva el resultado completo.

Por ejemplo el número 10 tiene el factorial $10*9*8*7*6*5*4*3*2*1 = 3628800$. Quitamos los ceros de la derecha para obtener 36288 finalmente nos piden 3 dígitos imprimimos 288.

Input

Los datos de entrada son los números N y K separados por un espacio donde $1 \leq N \leq 20$ y $1 \leq K \leq 9$. La entrada termina cuando no hay mas datos.

Output

Por cada dato de entrada escriba los dígitos especificados por K en una línea.

Ejemplo de entrada	**Ejemplo de salida**
10 3	288
6 1	2
6 3	72
7 2	04
20 9	200817664
1 1	1

Capítulo 5

Cadenas

5.1. Definición

Se dice cadenas a las secuencias de texto (caracteres) con las que se quiere trabajar. Todos estos textos generalmente están en ascii. Uno puede crear su propia representación, pero, para mostrar por pantalla debe estar en conjunto de caracteres definido en el sistema. En el español, es ascii, con extensiones para los caracteres latinos y se denomina *latin-1* en Windows y *iso-8859-15* en Linux y otros sistemas operativos. En java se denominan *Strings* y pertenecen a la clase *String*.

Como ve no es un tipo básico, es una clase. Por lo tanto, se define como clase y tienen métodos para trabajar con ella. La sintaxis de java es:

```
String nombre="cadena";
```

La definición se la comienza con el nombre de la clase que es *String*, luego el nombre de la variable y el contenido es el conjunto de caracteres encerrado entra comillas. Por ejemplo:

```
String saludo="Hola";
```

5.2. Recorrido de la cadena

Las cadenas se pueden ver como un conjunto de caracteres uno a continuación del otro. El nombre de la cadena representa la dirección de memoria donde se encuentran los caracteres que conforma la cadena. Por esta razón no es posible acceder directamente a sus valores y se requiere utilizar los métodos de la clase.

Consideremos la siguiente definición:

String saludo="Hola";
String saludo2=saludo;

Lo que se hizo primero es definir la cadena *saludo* que apunta al texto *Hola*, segundo se asigno a la dirección de la cadena "*Hola*" al nombre *saludo2*. Esto significa que solo existe un "*Hola*", cualquier cambio en *saludo* es un cambio en *saludo2*.

Ahora si comparamos *saludo* == *saludo2* el resultado es verdadero porque ambos apuntan a la misma cadena.

Si deseamos tener dos cadenas "*Hola*" es necesario definir como sigue:

```
String saludo="Hola";
String saludo2=new String("Hola");
```

Ahora si comparamos *saludo* == *saludo2* el resultado es falso porque ambos apuntan a diferentes cadenas.

Para recorrer una cadena se tiene el método *charAT(posicion)*. Vemos la cadena "HOLA".

	H	O	L	A
Posición	0	1	2	3

Para mostrar toda una cadena en pantalla tenemos la instrucción que vimos:

```
System.out.print(cadena)
```

Si queremos mostrar la misma, carácter por carácter, utilizamos *charAt*. Por ejemplo para mostrar la cadena *saludo* recorremos la cadena con un *for* como sigue:

```
String saludo="Hola";
for (int i = 0; i< 4; i++)
    System.out.print(saludo.charAt(i));
```

5.3. Métodos de la clase cadena

La clase cadena tiene varios métodos de los que describimos los más utilizados:

Método	Descripción	Resultado
charAt(int)	Devuelve el carácter ubicado en la posición	un carácter
compareTo(str)	Compara la cadena con *str*	0 cuando son iguales, negativo si es menor y positivo si es mayor
compareToIgnoreCase(str)	Compara la cadena con *str* tomando todas las letras como mayúsculas y minúsculas como iguales	0 cuando son iguales, negativo si es menor y positivo si es mayor
concat(str)	concatena la cadena con *str*	una cadena
contains(str)	Busca si existe la cadena *str*	verdadero o falso
endsWith(str)	Verifica si la cadena termina con *str*	verdadero o falso
equals(str)	Compara si dos cadenas son iguales	verdadero o falso
equalsIgnoreCase(str)	Compara si dos cadenas son iguales considerando las mayúsculas y minúsculas como iguales	verdadero o falso
indexOf(str)	Busca la posición donde comienza la cadena *str*	devuelve la posición o -1 si no existe
indexOf(str, int)	Busca después del índice *int* la posición donde comienza la cadena *str*	devuelve la posición o -1 si no existe
lastIndexOf(str)	Busca la última posición donde comienza la cadena *str*	devuelve la posición o -1 si no existe
lastIndexOf(str, int)	Busca después del índice *int* la última posición donde comienza la cadena *str*	devuelve la posición o -1 si no existe
length()	Retorna el tamaño de la cadena	un entero
replace(char1,char2)	Reemplaza todos los caracteres *char1* por *char2*	cadena
startsWith(str)	Indica si la cadena comienza con *str*	verdadero o falso
startsWith(str, int)	Indica si la cadena comienza con *str* después del índice *int*	verdadero o falso
substring(int)	Obtiene la cadena desde la posición hasta el final de la cadena *int*	cadena
substring(int, int)	Obtiene la cadena desde la posición *int* hasta la posición *int*	cadena
toLowerCase()	Convierte todas las letras a minúsculas	cadena
toUpperCase()	Convierte todas las letras a mayúsculas	cadena

Para ejemplificar el uso de los métodos descritos consideremos las siguientes definiciones:

```
String cad1 = "AbCdEf";
String cad2 = "aBcDeF";
String Cad3 = "abcabcabc";
```

- System.out.println(cad.charAt(0)); da *A*

- System.out.println(cad1.compareTo(cad2)); da -32
- System.out.println(cad1.compareToIgnoreCase(cad2)); da 0 son iguales
- System.out.println(cad1.concat(cad3)); da $AbCdEfabcabcabc$
- System.out.println(cad3.contains("ab")); da $true$
- System.out.println(cad3.endsWith("ab")); da $false$
- System.out.println(cad3.startsWith("ab")); da $true$
- System.out.println(cad3.indexOf("bc")); da 1
- System.out.println(cad3.indexOf("bc",3)); da 4
- System.out.println(cad3.substring(2,5)); da cab
- System.out.println(cad3.lastIndexOf("bc")); da 7
- System.out.println(cad3.replace('a','x')); da $xbcxbcxbc$
- System.out.println(cad1.toLowerCase()); da $abcdef$
- System.out.println(cad3.length()); da 9

5.4. Lectura del teclado

Utilizando la clase *Scanner* se tienen los siguientes métodos para cadenas

Método	Descripción
next()	Lee los siguientes caracteres delimitados por espacio y los almacena en una cadena
nextLine()	Lee toda una línea, no separa por espacios
hasNextLine()	Pregunta si hay una siguiente línea
hasNext()	Pregunta si hay mas caracteres en la entrada

Veamos algunos ejemplos considerando la entrada como sigue:

```
la casa
de la escuela
es
```

El siguiente código

```
Scanner lee = new Scanner(System.in);
String cad1 = lee.next();
String cad2 = lee.next();
String cad3 = lee.next();
```

hará que:

```
cad1 = "la";
cad2 = "casa";
cad3 = "de";
```

Si leemos por líneas:

```
Scanner lee = new Scanner(System.in);
String cad1 = lee.nextLine();
String cad2 = lee.nextLine();
String cad3 = lee.nextLine();
```

hará que:

```
cad1 = "la casa";
cad2 = "de la escuela";
cad3 = "es";
```

Ahora si

```
Scanner lee = new Scanner(System.in);
String cad1 = lee.next();
String cad2 = lee.next();
String cad3 = lee.nextLine();
```

hará que:

```
cad1 = "la";
cad2 = "casa";
cad3 = "";
```

La tercera lectura quedó en blanco porque lo que se hizo es leer hasta el final de la línea y no existían más caracteres. El método *nextLine()* lee desde la posición actual hasta el fin de linea. En cambio el método *next* salta los caracteres de fin de línea.

5.5. Convertir de cadena a Integer

Supongamos que hemos leído un número entero con el método *next()*. En este momento tenemos una variable de tipo *String* que contiene un número. Para convertir este a un número entero utilizamos la instrucción *Integer.parseInt(cadena)*. El resultado sera un número entero. La tabla siguiente muestra como convertir de cadena a diferentes tipos de datos.

Integer.parseInt	Convertir de cadena a tipo int
Double.parseDouble	Convertir de cadena a tipo double
Float.parseFloat	Convertir de cadena a tipo float
Long.parseLong	Convertir de cadena a tipo long

Si la cadena a convertir está en una base diferente a la decimal, por ejemplo en binario, podemos utilizar un segundo parámetro para especificar la base. Por ejemplo *Integer.parseInt("101",2)* da como resultado 5. De esta forma se puede incorporar un número en cualquier base.

Como ejemplo construyamos un programa para convertir el número 1234567 de base octal a decimal y binario. Para convertir a decimal escribimos el código siguiente:

```
int i = Integer.parseInt("1234567",8);
```

y para llevar a binario:

```
System.out.println(Integer.toBinaryString(i));
```

que puede verificar que produce los resultados 342391 en decimal y 1010011100101110111 en binario.

5.6. Manejo de excepciones

Que ocurre si el dato de entrada tiene un error, por ejemplo *Integer.parseInt("12ab")* o los caracteres no corresponden a la base especificada. El resultado será un error *NumberFormatException*. El lenguaje Java nos obliga a que el programa haga las previsiones necesaria para tratar con éstos y otros errores. Esto se denomina manejo de excepciones.

Existen dos tipos de formas de manejar las excepciones. La primera es la de decidir no hacer nada y hacer que el programa arroje el error. Esto se realiza colocando un claúsula *throws* en el método. Si se produce el error el java se encargara de mostrar el error. La sintaxis es como sigue:

```
public static void main(String[] args) throws NumberFormatException{
```

La segunda alternativa es que uno se preocupe de manejar el error, para esto, se utiliza una secuencia de instrucciones *try* y *catch*. La instrucción *try* especifica: intente hacer las instrucciones incluídas en el bloque, si da error pasa al bloque *catch*. La sintaxis es:

```
try {
    instrucciones
    }
catch (excepción variable) {
    manejo de error
    }
```

Expliquemos como controlar el error al convertir un número almacenado en una cadena a una variable entera. Considere el programa siguiente:

```
import java.util.Scanner;
public class TryCatch {
    public static void main(String[] args){
        Scanner lee = new Scanner(System.in);
        String cad1 = lee.nextLine();
         int i;
        try {
             i = Integer.parseInt(cad1);}
        catch (NumberFormatException n) {
            System.out.println(``Error "+n);
            i=0;
        }
        System.out.println(i);


    }
}
```

El flujo normal es cuando ingresa al bloque de instrucciones del *try*. El bloque de instrucciones del *catch* trata el error. Primero *catch (NumberFormatException n)* especifica el nombre de la excepción

que vamos a manejar. La variable n indica donde se recibirá el texto del mensaje de error. En el ejemplo del caso de error pusimos $i = 0$ y además mostramos el error.

5.7. Como procesar una línea de texto

En muchas ocasiones se quiere procesar una línea de texto, por ejemplo separar por palabras que vienen separadas por espacios. Este tipo de palabras se denominan *tokens*. Supongamos que tenemos *cad1 = "la escuela esta de fiesta"* para separar por palabras, podemos usar la clase *Scanner* otra vez. Definimos una variable de tipo *Scanner* y decimos que la entrada es *cad1* en lugar de *System.in*. Después usamos todos los métodos de la clase y podemos resolver el problema. El ejemplo siguiente muestra como podemos imprimir cada palabra de *cad1* en una línea.

```
import java.util.Scanner;
public class Divide {
    public static void main(String[] args) throws NumberFormatException{
        String cad1 = "la escuela esta de fiesta";
        Scanner lee = new Scanner(cad1);
         while (lee.hasNext()){
            System.out.println(lee.next());
            }


    }
}
```

5.8. Ejemplos de aplicación

1. Consideremos los celulares y los nombres que escribimos por el teclado. El teclado tiene 9 botones de los cuales tienen números, 8 tiene asignadas letras, como se muestra:

	ABC	DEF
GHI	JKL	MNO
PQRS	TUV	WXYZ

El problema consiste en leer un nombre e imprimir los números que se deben apretar para escribir este número. Para resolver el problema, requerimos un mecanismos para recorrer todos los elementos de la cadena ingresada, un sección de código para determinar a que número corresponde el carácter, y un proceso de concatenación para armar la respuesta.

La cadena la recorremos con un *for*, el límite se halla con el método *length()*, los caracteres se obtienen con *charAt*. Las instrucciones *if else* hallamos el número que le corresponde. La concatenación la hicimos con el operador +. El código que se muestra solo considera letras mayúsculas. Para que trabaje con letras minúsculas podemos utilizar el método *toUpperString*, para primero llevar todo a mayúsculas.

El código resultante es:

```
import java.util.Scanner;
public class Eje1 {
    public static void main(String[] args) {
```

```
        Scanner lee = new Scanner(System.in);
         while (lee.hasNext()){
            String nombre = lee.next();
            String resp="";
             for (int i=0; i <nombre.length(); i++){
                 if (nombre.charAt(i)<='C')
                     resp=resp+"2";
                 else
                     if (nombre.charAt(i)<='F')
                     resp=resp+"3";
                 else
                     if (nombre.charAt(i)<='I')
                     resp=resp+"4";
                 else
                     if (nombre.charAt(i)<='L')
                     resp=resp+"5";
                 else
                     if (nombre.charAt(i)<='L')
                     resp=resp+"6";
                 else
                     if (nombre.charAt(i)<='O')
                     resp=resp+"7";
                 else
                     if (nombre.charAt(i)<='S')
                     resp=resp+"8";
                 else
                     if (nombre.charAt(i)<='V')
                     resp=resp+"8";
                 else
                     if (nombre.charAt(i)<='Z')
                     resp=resp+"9";
                 }
             System.out.println(resp);
             }
    }
}
```

```
Ejemplo de entrada

UMSA

Ejemplo de salida
8672
```

2. Supongamos que queremos contar cuantas veces se repite la primera palabra en una línea de texto. Para esto leemos la primera palabra y guardamos ésta para comparar con todas las siguientes. Esto se hace con el método *equals*. Para saber si llegamos al final de la línea usamos el método *hasNext*. Veamos el código

```
import java.util.Scanner;
public class Eje2 {
    public static void main(String[] args) {
        Scanner lee = new Scanner(System.in);
```

```
        while (lee.hasNextLine()){
            int cont=1;
            String l=lee.nextLine();
            Scanner palabra = new Scanner(l);
            String p1=palabra.next();
            while (palabra.hasNext()){
                String p2=palabra.next();
                if (p1.equals(p2))
                    cont++;
            }
            System.out.println(cont);
        }
    }
}
```

En este programa hemos utilizado dos flujos de entrada con *Scanner*. El primer flujo se crea una sola vez para leer línea a línea del teclado. El segundo se crea cada vez que leemos una cadena para hacer recorrer todas las palabras de la línea. El contador comienza en uno que corresponde a la primera palabra.

```
Ejemplo de entrada

hola la hola cosa

Ejemplo de salida
2
```

3. Se quiere buscar el primer texto que diga "*error*", puede ser una palabra separada por espacios o no. La búsqueda se realiza con el método *indexOf* cuyo resultado es un entero que indica la posición donde se inicia la cadena. Cuando es negativo indica que no encuentra ésta cadena. El método *substring* permite obtener la primera parte del texto y también la segunda mitad. Concatenando ambas se obtiene una cadena donde se elimino la palabra "*error*".

```
import java.util.Scanner;
public class eje3 {
    public static void main(String[] args) {
        Scanner lee = new Scanner(System.in);
        while (lee.hasNextLine()){
            String l=lee.nextLine();
            int inicio=l.indexOf(''error");
            if (inicio>=0)
                l=l.substring(0,inicio)+l.substring(inicio+5);
            System.out.println(l);
        }
    }
}
```

```
Ejemplo de entrada

abcerrordef
```

```
Ejemplo de salida
abcdef
```

5.9. Ejercicios

1. Concursos por SMS
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Los números de los concursos donde uno envía mensajes de texto *sms* son de 4 dígitos.

   Una empresa desea contratar un número que sea acorde con su sigla, por ejemplo la Universidad Mayor de San Andrés desea el número *UMSA*. El registro que se tiene es por números y no por siglas. La compañía de celulares le ha pedido que haga un programa para determinar si una sigla está disponible.

   En su celular usted podrá observar que la asignación de letras es como sigue: $abc = 2$, $def = 3$, $ghi = 4$, $jkl = 5$, $mno = 6$, $pqrs = 7$, $tuv = 8$, $wxyz = 9$.

   Con esta información vemos que el número *UMSA* quedaría representados por 8672.

   ### Input

   La entrada consiste de varias líneas. La primera es la sigla de la cual queremos conocer si hay disponibilidad del número. Esto significa que éste número no fue asignado a otro concurso.

   La segunda contiene la cantidad de números asignados. Las siguientes los números asignados separados por espacios. La entrada termina cuando no hay más datos en la entrada.

   ### Output

   La salida consiste de una de la siguientes frases: *disponible*, cuando la sigla no existe en la lista de los números asignados y *No disponible* cuando la sigla existe en la lista de los números asignados.

Ejemplo de entrada	**Ejemplo de salida**
umsa 5 2828 3333 4587 2345 8672	No disponible

2. Dieta

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

El médico le ha dado su dieta, en que cada carácter (letra) corresponde a cierto alimento que debe comer. Se sabe también lo que ha comido para el desayuno y el almuerzo. Cada carácter corresponde a un tipo de alimento que, ha comido ese día.

Ha decidido que comerá todo el alimento restante de su dieta durante la cena.

Si usted ha cometido fraude de cualquier modo, o por comer demasiado de un tipo de alimento, o por comer cierto alimento que no está en su dieta, usted debe devolver la palabra *TRAMPOSO*.

Por ejemplo si su dieta es *ABCD* y el desayuno consumió *AB* en el almuerzo *C*. Aquí, ha comido la mayor parte de su alimento para el día. Así, puede comer sólo una *D* para la cena.

Input

La entrada consiste en múltiples casos de prueba. Cada caso de prueba contiene tres líneas. La primera línea contiene la dieta que contendrá entre 0 y 26 caracteres. Cada carácter en la dieta contendrá entre 0 y 26 caracteres y será una letra en mayúsculas $A - Z$, y no incluye caracteres duplicados.

La segunda línea contiene el desayuno. Cada carácter en el desayuno contendrá entre 0 y 26 caracteres y será una letra en mayúsculas $A - Z$, y no incluye caracteres duplicados.

La tercera línea es la del almuerzo. Cada carácter en el almuerzo contendrá entre 0 y 26 caracteres y será una letra en mayúsculas $A - Z$, y no incluye caracteres duplicados.

Ningún carácter aparecerá en ambos desayuno y almuerzo.

La entrada termina cuando no hay más datos en la entrada.

Output

Escribirá una línea de salida por cada caso de entrada con la cadena de caracteres que representa lo que puede consumir en la cena o la palabra *TRAMPOSO* si consumió alimentos que no están en su dieta.

Ejemplo de entrada	**Ejemplo de salida**
ABCD AB C ABEDCS "" "" EDSMB MSD A "" "" "" "IWANTSODER "SOW "RAT	D ABCDES TRAMPOSO "" DEIN

3. Cercas

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una sucesión de caracteres es llamada una cerca, si ésta consiste de símbolos | y − en forma alternada.

Empezando con cualquiera de ellos |−|−|−| o −|−|. Note que |−||−| o −− no son cercas, porque cada uno contiene símbolos iguales subsecuentes.

Dada una cadena s, encuentre la subcadena consecutiva más larga que sea una cerca y devuelva su longitud.

Por ejemplo la cadena |−|−−−|−|−−−|−| posee la subcadena −|−|− de caracteres en forma alterna que conforman la cerca más larga de la cadena cuya longitud es 5

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba en una línea. Cada línea contiene una secuencia de caracteres que son | o −. La cadena tendrá una longitud máxima de 50 caracteres.

La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba imprima una línea con el tamaño de la cerca más grande.

Ejemplo de entrada

```
|-|-|
-|-|-|-
||||||
|-||-|-
|-|---|-|---|-|
|||-||--|--|---|-||-|-|-|--||---||-||-||-|--||
```

Ejemplo de salida

```
5
7
1
4
5
8
```

4. **Mensajes ocultos**
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Algunos textos contienen mensajes ocultos. En el contexto de este problema el mensaje oculto de un texto se compone de la primera letra de cada palabra del texto en el orden en que aparecen.

   Una palabra es una secuencia consecutiva de letras. Puede haber varios espacios entre las palabras. Además, el texto puede contener varios espacios.

   Para hallar el mensaje oculto debe obtener la primera letra de cada palabra. Por ejemplo, si la cadena es *compete online design event rating*, tomando la primera letra de cada palabra la respuesta es *coder*.

   Dado una cadena de texto que consta de sólo letras minúsculas y espacios entre ellas, halle el mensaje oculto.

Input

La entrada consiste de varios casos de prueba, cada uno en una línea. Cada línea contiene solo letras minúsculas $a - z$ y espacio y serán de 1 a 50 caracteres. La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada imprima el mensaje oculto en una línea. Si la línea no tiene caracteres debe devolver una línea en blanco.

Ejemplo de entrada

Ejemplo de entrada

```
compete online design event rating
round  elimination during  onsite  contest
```

Ejemplo de salida

```
coder
redoc
```

5. Escanear documento
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted ha sido encargado de escribir una función que escaneará un documento, del cual determinará cuántas veces aparece una determinada palabra o frase en ese documento. Sin embargo, es importante que su solución no tome en cuenta palabras que se superponen. Por ejemplo, si el documento es *abababa*, y la palabra clave de búsqueda es *ababa*, usted podría encontrar la palabra clave a partir del índice 0 o en el índice 2 pero no ambos. Ya que la primera cadena que empieza en 0 incluye a la otra que empieza en 2.

Para encontrar el mayor conjunto de palabras que no se solapan, realice el siguiente procedimiento: empezando desde la izquierda, encontrar la primera aparición de la cadena de búsqueda, entonces, continuando con el carácter inmediatamente después de la cadena de búsqueda seguir buscando la siguiente aparición. Repita hasta que no haya nuevas ocurrencias. Al continuar inmediatamente después de cada ocurrencia encontrada garantizamos que no vamos a contar con más superposiciones.

En las cadenas de la entrada debe ignorar los espacios.

Input

La entrada consiste de varios casos de prueba, cada uno en dos líneas. La primera línea contiene la cadena que no tendrá más de 50 caracteres. La segunda línea la palabra a buscar. La palabra a buscar no contendrá espacios.

La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba escriba una línea con el número de ocurrencias sin solapar que existen de la palabra buscada en la cadena.

Ejemplo de entrada	**Ejemplo de salida**

`abababababa`
`ababa`
`abababababa`
`aba`
`abcdefghijklmnop qrstuvwxyz`
`pqrs`
`aa aa a a`
`aa`
	1
	2
	1
	3

6. Esperanto

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números son mucho más fáciles de escribir en esperanto que en español. Los números del 1 al 10 se detallan como sigue: *unu*, *du*, *tri*, *kvar*, *Kvin*, *ses*, *Sep*, *OK*, *Nau*, *dek*. Números del 11 al 19 se escriben: *dek unu*, *dek du*, ..., *dek Nau*, un *dek* seguido de un solo espacio y el nombre del último dígito. Números 20 al 29 se escriben: *dudek*, *dudek unu*, *dudek du*, ..., *dudek Nau*. Del mismo modo, 30 es *tridek*, ..., 90 es *Naudek*. Sólo se unen el número de decena y *dek*. No hay excepciones como *doce* o *quince* en español.

Input

La entrada consiste en varios casos de prueba. Cada caso de prueba es un número entero n, $1 \leq n \leq 99$ en una línea. La entrada termina cuando no hay más datos en la entrada.

Output

Por cada caso de prueba en la entrada escriba una línea con el nombre en esperanto.

Ejemplo de entrada	**Ejemplo de salida**
1	unu
90	Naudek
11	dek unu
77	Sepdek Sep

7. Palindromes

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una palabra se dice palíndromo si la misma palabra es exactamente idéntica al derecho y al revés como por ejemplo ABA, AA, A y $CASAC$. Por otra parte cualquier secuencia de caracteres puede ser vista como 1 o más secuencias de palíndromos concatenadas. Por ejemplo *guapa* es la concatenación de los palíndromos *g*, *u* y *apa*; o *casacolorada* es la concatenación de *casac*, *olo*, *r*, *ada*. El problema consiste dado una palabra de a lo máximo 2000 letras minúsculas, imprima el número mínimo de palabras palíndromas en que se puede dividir.

Input

La entrada consiste de una línea por caso, cada línea contiene una cadena s de entre 1 y 2000 caracteres. La entrada termina con EOF. Puede estar seguro que la entrada no contiene más de 1000 casos.

Output

Por cada caso imprime el número mínimo de palabras palíndromas en que se puede dividir s.

Ejemplo de entrada	Ejemplo de salida
casacolorada	4
casac	1
hola	4

8. Contar Caracteres
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En este problema le dan una cadena de caracteres, y al final le preguntan cuantas veces aparecen ciertos caracteres.

Notas de los casos de prueba:

- Recuerde que en las pregunta puede estar el carácter " (espacio en blanco). Si existiera una pregunta por el carácter espacio en blanco en el primer caso de entrada, la respuesta sería 3 (suponiendo que no hay mas espacios al final de la oración).
- El carácter A es distinto que el carácter a.

Input

Para cada caso de prueba recibirás una cadena de caracteres la cual puede contener espacios y no excede mas de 1001 caracteres, en la siguiente línea recibirá un entero Q el número de preguntas, en las siguientes Q líneas recibirá en cada una de ellas un carácter.

Los casos de prueba terminan cuando encuentre un #

Output

Para cada caso de prueba debe imprimir Q líneas, en cada una el número de veces que aparece el carácter de la pregunta. Después de cada caso de prueba debe imprimir un línea en blanco.

Ejemplo de entrada	**Ejemplo de salida**
Hola Cara de bola 2 H a Ahi viene el Coco 3 c C a #	1 4 1 1 0

9. Tapices de Colores
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A Taro le gustan las cosas de colores, especialmente los tapices.

El cuarto de Taro está dividido en L tapices cuadrados organizados en una línea. Cada tapiz es de uno de los siguientes colores: rojo, verde, azul o amarillo representados por R, G, B y Y respectivamente. A usted le dan un cadena representando los tapices del cuarto. El carácter i del cuarto representa el color del tapiz.

Ha decidido cambiar el color de algunos tapices de tal forma que dos tapices adyacentes no tengan el mismo color.

Le piden hallar el mínimo número de tapices que hay que cambiar.

Por ejemplo:

Si la entrada fuera *RRRRRR* cambiamos a *RGRGRG* y la respuesta es cambiar 3.

Si la entrada fuera *BBBYYYYYY* la respuesta es 4, porque podemos cambiar a *BRBYRYRYR*.

Input

La primera línea indica cuantos casos de prueba hay. Las siguientes líneas tienen un caso de prueba por línea. Cada línea es una cadena con un máximo de 10 caracteres representando los colores de los tapices.

Output

La salida es el número de tapices que hay que cambiar. Se imprime por cada caso de prueba un número en una línea.

Ejemplo de entrada	**Ejemplo de salida**
5 RRRRRR GGGGGGG BBBYYYYYY BRYGYBGRYR RGGBBBRYYB	3 3 4 0 3

10. Código Rojo
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Estamos buscando las repeticiones de la palabra *red* en un pedazo de texto. Por ejemplo en el texto *the detective questioned his credibility* encontramos la palabra *red* en la palabra *credibility.*

No nos interesa si non mayúsculas o minúsculas. Así que en la frase *Server ERRED in Redirecting Spam* encontramos *red* en *ERRED* y tambien en *Red.*

Dada una oración cadena se desea conocer cuántas veces aparece la palabra *red.*

Input

Existen varios casos de prueba. Cada caso de prueba viene en una línea. El texto estará entre 1 y 50 caracteres. Solo contiene las letras $(A-Z, a-z)$ y espacios.

Output

Por cada caso de prueba imprima una línea con el número de veces que aparece *red.* La entrada termina cuando no hay más datos.

Ejemplo de entrada

```
the detective questioned his credibility
Server ERRED in Redirecting Spam
  read the RED sign   said fReD
pure delight
Note that re d is not counted as an occurrence of red.
```

Ejemplo de salida

```
1
2
2
0
1
```

11. Cifrado por Desplazamiento
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted ha decidido crear sus propio método de cifrado de datos para cadenas que contienen letras minúsculas y espacios. Usted comienza dividiendo el alfabeto en dos grupos. El primero consiste de $primerTam$ y el restante consiste de las $26 - primerTam$ letras. Para cifrar el mensaje realiza lo siguiente:

- Si es un espacio se deja como está.
- Si es una letra del primer grupo, es movida $primerRote$ hacia adelante volviendo al principio si es necesario. Por ejemplo si $primerTam = 6$ y $primerRote = 2$ la letra A se convertirá en una C y la F en una B.
- Si una letra pertenece al segundo grupo se moverá $segundoRote$ letras adelante en el grupo, rotando si es necesario.

Dados $primerTam$, $primerRote$, $segundoRote$ y el mensaje, imprima el mensaje cifrado en una línea.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba está en una sola línea y contiene tres números enteros $1 \leq primerTam \leq 25$, $0 \leq primerRote \leq primerTam - 1$, $0 \leq segundoRote \leq 25$, y el mensaje entre 1 y 50 caracteres inclusive. La entrada termina cuando no hay más datos.

Output

Por cada caso de entrada imprima una línea con el mensaje cifrado.

Ejemplo de entrada

```
13 0 0 this string will not change at all
13 7 0 only the letters a to m in this string change
9 0 16 j to z will change here
17 9 5 the quick brown fox jumped over the lazy dog
3 1 2  watch   out for strange  spacing
```

Ejemplo de salida

```
this string will not change at all
onfy tbl fltt1rs h to g cn tbcs strcna jbhnal
z sn y vikk chamge heqe
yqn izalc kwgsf ogt bzehnm grnw yqn djvu mgp
ybvaj   qwv hqt uvtbpig  urbakpi
```

12. Quitar puntuación
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A algunos escritores les gusta super enfatizar ciertos puntos utilizando múltiples exclamaciones en lugar de una. Por ejemplo *que Bárbaro!!!!!*. Otras veces expresan su sorpresa con múltiples exclamaciones e interrogaciones, por ejemplo *Realmente te gusta!?!?!?!?*.

Usted está editando un documento para su publicación, y quiere eliminar la puntuación extra. Cuando ve varios signos de exclamación seguidos los remplaza por uno solo. Si ve un conjunto de interrogaciones con uno o ninguna exclamación remplazarlos por una interrogación simple. Si ve exclamaciones e interrogaciones combinadas remplazar por una interrogación. Vea los datos de entrada y salida para mayores ejemplos

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba viene en una línea conteniendo el texto a procesar. Tendrá entre 1 y 50 caracteres inclusive, letras mayúsculas, minúsculas, espacio y signos de puntuación, exclamación e interrogación.

La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada imprima la cadena modificada de acuerdo a la explicación.

Ejemplo de entrada

```
"This cheese is really great!!!!!"
"You really like THIS cheese!?!?!?!!!?"
" !!?X! ?? This is delicious!!! ??!a!?!"
```

Ejemplo de salida

```
"This cheese is really great!"
"You really like THIS cheese?"
" ?X! ? This is delicious! ?a?"
```

13. Flecha más Larga

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En este problema una flecha a la izquierda se define como el carácter < seguido inmediatamente de cero o más caracteres −. Un flecha doble a la izquierda se define como un carácter < seguido de cero o más caracteres consecutivos =.

Una flecha a la derecha se define como cero o más caracteres − seguidos del carácter >. Un flecha doble a la derecha se define como cero o más caracteres = seguidos del carácter >.

Dada una cadena se quiere hallar la longitud de la flecha más larga.

Input

La entrada contiene varios casos de prueba. Cada caso de prueba es una cadena en una línea. La cadena contiene entre 1 y 50 caracteres. Cada carácter será <, >, −, =.

La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba escriba en una línea la longitud de la cadena más larga. Si no existe una cadena escriba −1.

Ejemplo de entrada	Ejemplo de salida
<---><--==> <<<<<<<<< ----==- <----=====>	4 1 -1 6

14. Cadenas Palíndromes
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una cadena es un palíndrome cuando leída de derecha a izquierda o de izquierda a derecha es la misma. Juan tiene dos cadenas A y B y está curioso de conocer como introducir la cadena B en la cadena A para que el resultado sea una cadena palíndrome. Usted ha acordado ayudar en determinar cuantas variantes existen para formar palíndromes. Si insertamos la cadena B en diferentes lugares se considera una variante.

Por ejemplo sea $A = aba$ y $B = b$ B se puede colocar en 4 lugares diferentes:

- Antes del primer caracter. Dando *baba* que no es palíndrome.
- Después de la primera a obtenemos *abba* que es palíndrome.
- Después de la b, dando *abba* que es palíndrome
- Después de la última a, dando *abab*, no es palíndrome

Para este ejemplo el existen dos variantes.

Input

Existen varios casos de prueba. Cada caso de prueba consiste de una línea con las cadenas A, B separados por un espacio. Las cadenas tienen entre 1 y 50 caracteres inclusive. La entrada termina cuando no hay más datos.

Output

Por cada caso de entrada escriba el número de variantes que hay para construir cadenas palíndromes.

Ejemplo de entrada	Ejemplo de salida
`aba b`	2
`aa a`	3
`aca bb`	0
`abba abba`	3
`topcoder coder`	0

15. Sumando

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Jaime es un estudiante muy aplicado en su segundo año de la escuela. Recientemente decidió convertir sus apuntes a una versión electrónica. Pero encontró que sus apuntes estaban llenos de manchas de tinta.

Utilizó un escaner y mando todo por un paquete de OCR, si lo codificó solo a la edad de 8 años. El paquete de OCR remplazaba todas las manchas de tinta por la palabra *machula*.

Las notas de Jaime consisten en ejercicios simples de matemáticas, que son la adición de dos números enteros. Su tarea es recuperar la parte dañada del documento.

Input

La entrada consiste en varios casos de prueba. La primera línea contiene el número de casos de prueba. Las siguientes líneas corresponden a un caso de prueba en cada línea.

Cada caso de prueba representa una ecuación de la forma *número + número = número*, donde cada número es positivo. Una parte de la ecuación es remplazada por la palabra *machula*. La cadena siempre cubre una secuencia consecutiva de dígitos.

La entrada termina cuando no hay más datos.

Output

La salida es una línea por caso de prueba con la ecuación correcta, donde hemos remplazado la palabra *machula* por los dígitos correctos.

Ejemplo de entrada	**Ejemplo de salida**
3 23 + 47 = machula 3247 + 5machula2 = 3749 machula13 + 75425 = 77038	23 + 47 = 70 3247 + 502 = 3749 1613 + 75425 = 77038

16. Oración Baile

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una oración se denomina baile si su primera letra es mayúscula y el caso del siguiente caracter o letra posterior es lo contrario al caracter anterior. Los espacios deben ser ignorados al determinar el caso de una letra. Por ejemplo, "A b Cd "es una oración baile porque la primera letra es ('A') es en mayúsculas, la siguiente letra ('b') es minúscula, la siguiente letra ('C') es en mayúsculas, y la siguiente letra ('d') es minúscula. Se le dará una oración como una cadena. Cambie la oración en una frase baile para esto cambie los casos de las letras cuando sea necesario. Todos los espacios en la frase original deben ser preservados.

Input

La entrada de datos consiste en una oración larga, el mismo puede contener entre $[1, 50]$ caracteres, además cada caracter será una letra de ('A'-'Z', 'a'-'z') o un espacio en blanco ''. La entrada termina cuando no hay más datos.

Output

Escriba para cada caso de prueba la oración baile correspondiente a la entrada. Cada oración debe imprimirse en una línea.

Ejemplo de entrada	**Ejemplo de salida**

`This is a dancing sentence`
` This is a dancing sentence`
` z`
`  Aa`
	`ThIs Is A dAnCiNg SeNtEnCe`
	` ThIs Is A dAnCiNg SeNtEnCe`
	` Z`
	`  Aa`

17. Buscador de Diamantes

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted esta encomendado de buscar diamantes en una mina peculiar. Esta mina es una cadena de caracteres que tienen los símbolos '<'y '>', el diamante es representado por la subcadena de forma "<>". Cada vez que encuentre un diamante debe extraer el mismo y la mina residual se actualiza mediante la eliminación de los 2 caracteres de la cadena. Por ejemplo, si usted tiene una mina como "><<><>>><", puede empezar por quitar la primera aparición de "<>"para obtener "><<>>><", luego retire el único diamante restantes para obtener "><>><". Tenga en cuenta que esto produce un nuevo diamante al cual puede quitar para obtener ">><". Dado que no existen diamantes izquierda, su expedición termina.

Dado una mina de representada en una cadena, devolver el número de diamantes que se pueden encontrar. Tenga en cuenta que el orden en que se quitan apariciones simultáneas de los diamantes es irrelevante; cualquier orden dará lugar al mismo resultado.

Input

La entrada de datos consiste en una cadena que representa la mina, esta debe contener entre $[1 - 50]$ caracteres cada carácter de la mina será '<' o '>'. La entrada termina cuando no hay más datos.

Output

Escriba para cada caso de prueba el número de diamantes que se pueden extraer de la mina.

Ejemplo de entrada	**Ejemplo de salida**
><<><>>><	3
>>>><<	0
<<<<<<<<<>>>>>>>>>	9
><<><<>>>><<>><<><<>><<<>>>>>><<<	14

18. Descifrando

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Julio César utilizó un sistema de criptografía, ahora conocido como cifrado de César, que cambia cada letra por la siguiente letra que este 2 posiciones más a través del alfabeto (por ejemplo, 'A', se cambia por 'C'y 'R'cambia a 'T'). Si la letra a cifrar está al final del alfabeto el recorrrido da una vuelta o sea vuelve al principio, por ejemplo 'Y'se cambia por la letra 'A'. Es posible cambiar el desplazamiento de las posiciones por cualquier número. Dado un texto codificado y un número de desplazamiento de las letras, devolver el mensaje decodificado.

Por ejemplo el mensaje, "TOPCODER"utilizando dos posiciones de desplazamiento se codifican como "VQREQFGT". Entonces, si se da el mensaje codificado "VQREQFGT"y 2 de desplazamiento como entrada, hallar el mensaje decodificado o sea mostrar "TOPCODER".

Input

La entrada de datos consiste en el mensaje codificado y N número de desplazamiento. El mensaje debe contener entre $[1 - 50]$ caracteres cada caracter es una letra mayúscula del alfabeto 'A'-'Z'. N es entero y esta entre $0 \leq N \leq 25$ La entrada termina cuando no hay mas datos.

Output

Muestre para cada caso de prueba el mensaje decodificado es decir el mensaje original.

Ejemplo de entrada	**Ejemplo de salida**
VQREQFGT 2 ABCDEFGHIJKLMNOPQRSTUVWXYZ 10 TOPCODER 0 ZWBGLZ 25	TOPCODER QRSTUVWXYZABCDEFGHIJKLMNOP TOPCODER AXCHMA

19. Fácil SQL

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

SQL (Structured Query Language, Lenguaje Estructurado de Consulta) es un lenguaje hecho para manipular los datos contenidos dentro de una Base de Datos. Cada tabla en una Base de Datos tiene un DDL (Data Definition Language, Lenguaje de Definición de Datos) y un DML (Data Management Language, Lenguaje de Manipulación de Datos).

Para hacer la Manipulación de Datos más fácil, queremos crear la sentencia $SELECT$, usando la Definición de Datos de cada tabla, que usa la sentencia $CREATE\ TABLE$.

Por ejemplo, tenemos la tabla: $CREATE\ TABLE\ table1\ (field1\ type1,\ field2\ type2)$ Y para obtener su contenido usamos: $SELECT\ field1,\ field2\ FROM\ table1$

Tu tarea es dada una sentencia $CREATETABLE$, construir la sentencia $SELECT$.

Input

La entrada tiene varios casos de prueba. Cada línea contiene una sentencia $CREATE\ TABLE$. El fin de la entrada es representada por un #.

Output

Para cada sentencia $CREATE\ TABLE$, imprimir la sentencia $SELECT$

Ejemplo de entrada

```
CREATE TABLE table1 (field1 type1, field2 type2)
CREATE TABLE table2 (field3 type3, field4 type4, field5 type5, field6 type6)
#
```

Ejemplo de salida

```
SELECT field1, field2 FROM table1
SELECT field3, field4, field5, field6 FROM table2
```

20. Tautogramas

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A Fiona siempre le gusto la poesía. Recientemente descubrió una forma fascinante de formas poéticas, que las denomino *Tautogramas*. Este es un caso especial de literatura donde las palabras adyacentes comienzan con la misma letra.

En particular una oración es un tautograma si todas sus palabras comienzan con la misma letra.

Por ejemplo las siguientes frases son tautogramas.

- Flowers Flourish from France
- Sam Simmonds speaks softly
- Peter pIckEd pePPers
- truly tautograms triumph

Fiona quiere que le ayude a determinar si las frases que escribió son tautogramas.

Input

Cada caso de prueba está dado en una sola línea que contiene una secuencias de hasta 50 palabras separadas por un espacio. Una palabra es una secuencia de hasta 20 caracteres contiguos con letras mayúsculas y minúsculas, del alfabeto inglés. El último caso de prueba esta seguido por una línea que contiene un solo '*'.

Output

Para cada caso escriba una línea conteniendo una 'Yén mayúsculas o una 'N'mayúscula en otros casos.

Ejemplo de entrada	Ejemplo de salida

`Flowers Flourish from France`
`Sam Simmonds speaks softly`
`Peter pIckEd pePPers`
`truly tautograms triumph`
`this is NOT a tautogram`
`*`
	`Y`
	`Y`
	`Y`
	`Y`
	`N`

21. Convertir las Propiedades de CSS
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los nombres de propiedades CSS son normalmente en minúsculas y escritos en una notación separadas por guiones, lo que significa que cada par de palabras adyacentes están separadas por un único guión. Por ejemplo *z-index, padding-left* , y *border-collapse*, son nombres típicos. Sin embargo, si usted desea utilizar JavaScript para establecer las propiedades de estilo CSS, necesita utilizar la notación sin guiones, donde cada palabra, excepto la primera comienza con una letra mayúscula y las palabras adyacentes no están separados por guiones. Todas las demás letras en minúsculas. Por ejemplo, *z-index* se convertirá en *zIndex* en la notación nueva.

Dado el nombre de una propiedad de CSS en una propiedad escrita en notación separadas por guiones. Convertir a la notación sin guiones.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba consiste de una cadena de entre $[1 - 50]$ caracteres, que solo contendrán letras minúsculas y guiones. La entrada termina cuando no hay más datos.

Output

Para cada caso de entrada mostrar la propiedad en la notación sin guiones.

Ejemplo de entrada	Ejemplo de salida

`z-index`
`border-collapse`
`top-border-width`
	`zIndex`
	`borderCollapse`
	`topBorderWidth`

Capítulo 6

Arreglos unidimensionales - vectores

6.1. Definición

Un vector o arreglo es una estructura de datos que permite almacenar valores secuencialmente en la memoria de la computadora. Se utiliza como contenedor para almacenar datos, en lugar de definir una variable por cada dato.

Todos los datos deben ser del mismo tipo. No se pueden mezclar tipos de datos.Los valores se colocan en secuencia a partir de la posición cero.

La sintaxis para definir un vector es:

1. Se define el tipo de dato seguido de [] que indica que se trata de un vector.
2. Se define el nombre de la variable.
3. Se coloca el símbolo igual seguido de *new* y el tipo de dato.
4. Finalmente se coloca [] con el tamaño definido entre ambos corchetes.

Veamos algunos ejemplos:

```
int [] vector = new int[8];
```

Esta definición crea una estructura que define 8 elementos enteros contiguos. El nombre *vector* es la dirección donde se encuentran los valores de la estructura de datos.

La figura 6.3 muestra como ha sido definido el vector.

Si definimos un vector de cadenas, la sintaxis es:

```
String [] vector = new String[8];
```

En este caso, cada elemento del vector es la dirección de la memoria donde se encuentra la cadena.

La figura 6.2 muestra como ha sido definido el vector de cadenas. Para definir un vector con valores iniciales el procedimiento es como sigue:

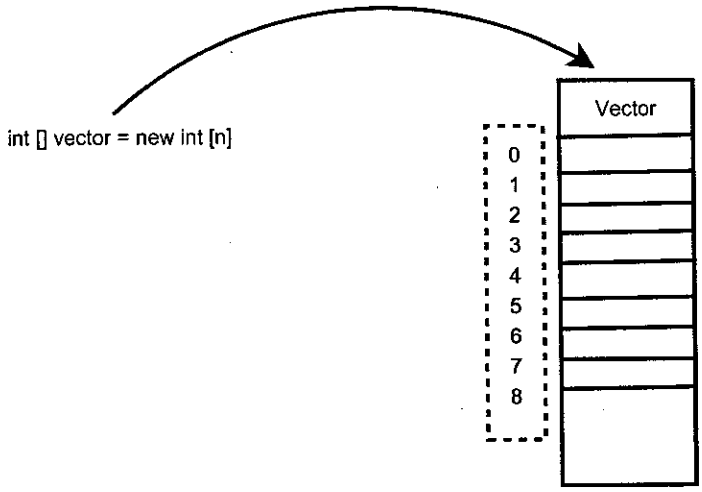


Figura 6.1: Definición de un vector

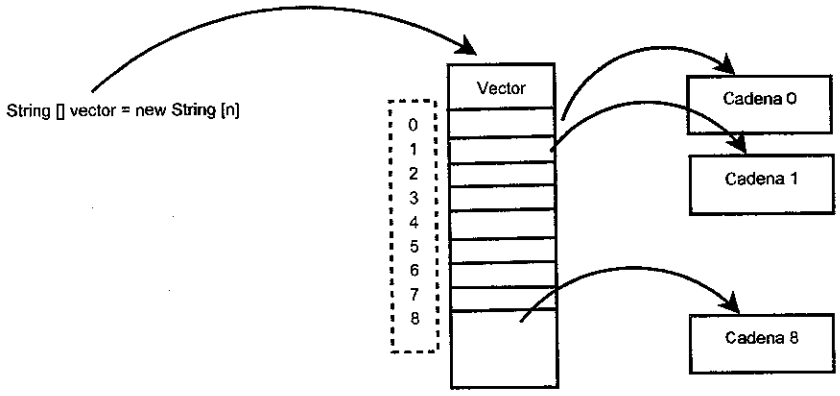


Figura 6.2: Definición de un vector de cadenas

1. Se define el tipo de dato seguido de [] que indica que se trata de un vector
2. Se define el nombre de la variable
3. Se coloca el símbolo igual seguido de un {
4. Se colocan todos los valores separados por una coma
5. Se termina con }

Como ejemplo definamos un vector de enteros:

$$\text{int [] vector} = \{1,10,5,15\};$$

Cuando los valores de un vector se encuentra en una cadena se utiliza el método *split* de la clase *String*. Por ejemplo:

```
String primos = "2, 3, 5, 7, 11";
int p= primos.split(",");
```

Acá el método *split* indica que hay múltiples valores delimitados por una coma.

6.2. Recorrido

Cada uno de los elementos de un vector se acceden por la posición donde está almacenado el valor que queremos acceder. Consideremos la definición:

$$\text{int [] v} = \{1,10,5,15\};$$

Para acceder a cualquier valor se escribe el nombre de la variable seguido de [] con el número de elemento que queremos acceder. Veamos $v[0] = 1$, $v[3] = 15$.

El tamaño del vector se almacena en la propiedad *length*. Note que no utilizamos los paréntesis en en las propiedades, en los vectores no es un *método*.

Para listar los elementos del vector definido, podemos utilizar un *for*

```
for (int i =0; i< v.length;i++)
     System.out.println(v[i]);
```

Los índices del vector van desde 0 hasta $length - 1$. La cantidad de valores es *length*. No es posible acceder a valores fuera de los límites de un vector.

Una segunda forma de recorrer un vector es utilizando la instrucción *for each*. La sintaxis es como sigue:

```
for (tipo variable: vector) {
     instrucciones
     }
```

Lo que indica es que se quiere recorrer todos los elementos de un vector y cada uno de los elementos será colocado en la variable, iteración por iteración. En código del ejemplo anterior es:

```
for (int i: v)
     System.out.println(i);
```

Como ve no tuvimos que hacer una referencia individual a cada elemento del vector.

6.3. Valores iniciales

Cuando definimos un vector los valores iniciales que tiene son:

- Cero cuando son valores numéricos
- Si son referencias como el caso de las cadenas toma el valor *null*
- Si son valores *boolean* toma el valor *false*

6.4. Ejemplos de aplicaciones

Para ejemplificar el uso de vectores vamos a hallar algunos indicadores estadísticos típicos: media, varianza, moda, máximo.

1. Hallar la media. La fórmula de la media es:

$$m = \frac{\sum_{i=1}^{n} x_i}{n}$$

Definiremos un vector para almacenar n valores, el recorrido del mismo será de 0 hasta $n - a$ inclusive. Para este los ejemplos siguientes colocaremos un vector con valores constantes. El código es como sigue

```
import java.util.Scanner;
public class Media {
    public static void main(String[] args) {
        int [] x= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8,};
        double m=0.0;
        int suma=0;
        for (int i=0; i<x.length;i++){
            suma+=x[i];
        }
        m=(double)suma/x.length;
        System.out.println(m);
    }
}
```

2. Hallar la varianza. La varianza se calcula con la fórmula:

$$v^2 = \frac{\sum_{i=1}^{n} (m - x_i)^2}{n}$$

Para resolver éste problema debemos hacer recorrer dos veces el vector una vez para hallar la media m y la segunda para hallar la varianza. El programa es como sigue:

```
import java.util.Scanner;
public class programa2 {
    public static void main(String[] args) {
        int [] x= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8,};
        double m=0.0;
        double v=0.0;
```

```
        int suma=0;
        for (int i=0; i<x.length;i++){
            suma+=x[i];
        }
        m=(double)suma/x.length;
        suma=0;
        System.out.println(''Media = "+m);
        for (int i=0; i<x.length;i++){
            suma+=(m-x[i])*(m-x[i]);
        }
        v=Math.sqrt(suma)/x.length;
        System.out.println(''Varianza = "+v);
    }
}
```

3. Hallar el máximo. Para hallar el valor máximo, primero definimos el máximo con el valor más pequeño que se puede almacenar. Luego debemos recorrer el vector y cada vez comparar con el máximo y almacenar el valor mayor. El siguiente programa halla el máximo:

```
import java.util.Scanner;
public class Maximo {
    public static void main(String[] args) {
        int [] x= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8,};
        int max=Integer.MIN_VALUE;
        for (int i=0; i<x.length;i++){
            max=Math.max(max,x[i]);
        }
        System.out.println(''MÃ¡ximo = "+max);
    }
}
```

4. Hallar la moda. La moda se define como el valor que más se repite, para esto haremos dos procesos. El primero para hallar las frecuencias de repetición de cada uno de los elementos de x. La segunda para hallar el máximo de esta repeticiones.

   Para hallar las frecuencia, es decir, cuantas veces ocurre cada elemento de x definimos un vector f donde en $f[0]$ servirá para contar cuantas veces aparece el 0, $f[1]$ para el 1, así sucesivamente. Hemos definido f de tamaño 10 porque sabemos que ningún valor excede a 10 valores. La instrucción $f[x[i]]++$ incrementa el vector de frecuencias en cada iteración.

   La segunda parte es la que halla el máximo. Con el mismo algoritmo anterior, almacenando junto al máximo la posición donde se encuentra. La posición representa la moda. En el segundo ciclo el recorrido es sobre el tamaño de f. El programa resultante es:

```
import java.util.Scanner;
public class Moda {
    public static void main(String[] args) {
        int [] x= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8,};
        int [] f= new int [10];
        for (int i=0; i<x.length;i++){
            f[x[i]]++;
        }
```

```
        int max=Integer.MIN_VALUE;
        int moda=0;
        for (int i=0; i<10;i++){
            if(f[i]>max){
                max=f[i];
                moda=i;
            }
        }
        System.out.println(''Moda = "+moda);
    }
}
```

En este ejemplo no es posible utilizar *Math.max* porque hay que almacenar dos valores.

5. Veamos un ejemplo más complejo. Se trata de sumar los puntos de una partida de bolos (bowling). En un juego de bolos un jugador tiene 10 oportunidades de anotar en el transcurso de un juego. Cuando lanza la bola, puede voltear 10 palos si voltea todos. En este caso se contabiliza 10 más lo que haya obtenido en sus siguientes dos jugadas. Cuando no voltea todos los palos, se anota la cantidad de palos, luego se hace un segundo lanzamiento y se agrega la cantidad de palos que volteo. Si en ésta segunda oportunidad completa los 10 palos volteados, se agrega lo que obtenga en su siguiente entrada.

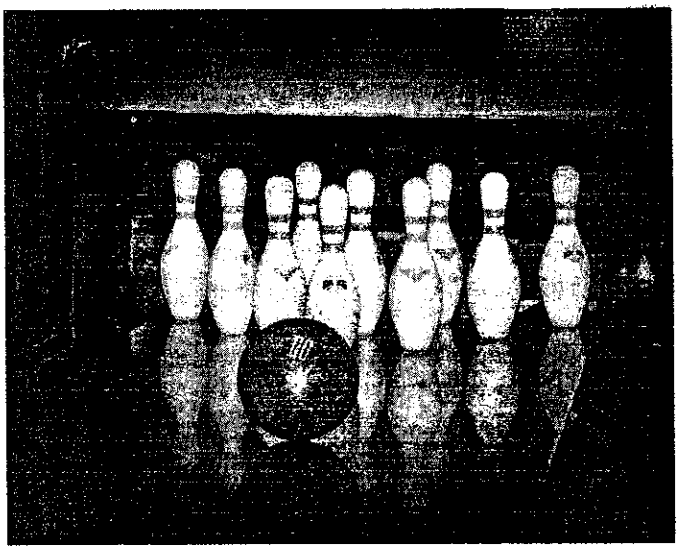


Figura 6.3: Juego de bolos

Por ejemplo $5, 2, 3, 4$ significa que primero volteo 5 palos, luego 2. En ésta entrada tiene 7 puntos en su segunda entrada tiene $3+4 = 7$ puntos. Veamos otro caso $10, 5, 2$ en este caso en la primera entrada volteo los 10 palos, por lo que el puntaje para esta entrada es $10 + 5 + 2 = 17$. En la siguiente entrada tiene $5 + 2 = 7$ acumulando hasta este momento 24 puntos. Si los datos son $7, 3, 5, 2$, significa que en su primera entrada volteo los 10 palos en dos intentos, por lo que el puntajes que se asigna es $7 + 3 + 5 = 15$, y no acumula los últimos 2 puntos. Dada la lista de todos los palos volteados en un juego, calcular el puntaje final.

Expliquemos el programa siguiente:

```
import java.util.Arrays;
import java.util.Scanner;
public class programa2 {
    public static void main(String[] args) {
        int [] x= {5,2,10,9,0,8,1,5,5,10,6,4,4,2,9,1,10,10,10};
        //int [] x= {10,10,10,10,10,10,10,10,10,10,10,10};
        int [] puntosJugada= new int [10];
        int jugadas=0, i=0;
        while (jugadas <10){
            if (x[i]==10){
                puntosJugada[jugadas]+=x[i+2]+x[i+1]+10;
                i++;
                jugadas++;
            }
            else
                if ((x[i]+x[i+1])<10){
                puntosJugada[jugadas]+=x[i]+x[i+1];
                i=i+2;
                jugadas++;
            }
            else {
                puntosJugada[jugadas]+=x[i+2]+10;
                i=i+2;
                jugadas++;
            }
        }
        int suma=0;
        for (int j: puntosJugada)
            suma+=j;
        System.out.println(''Datos iniciales "+Arrays.toString(x));
        System.out.println(''Puntos por jugada "+Arrays.toString(puntosJugada));
        System.out.println(''Puntaje final "+suma);
    }
}
```

Primero hemos definido un vector donde se establece el puntaje correspondiente a cada jugada, lo denominamos *puntosJugada*, de tamaño 10 dado que solò se puede jugar 10 veces. También establecemos un índice *jugada* que se incrementa en uno por cada jugada.

Se compara primero si derribó 10 palos, en cuyo caso se suma los dos valores siguientes y se incrementa el índice *jugada* en uno, y el índice que recorre el vector en uno. Luego vemos si hizo 10 en dos lanzamientos, en cuyo caso se incrementa el valor del próximo lanzamiento, y se incrementa el contador que recorre el vector en dos. En otros casos se incrementa el número de palos derribados.

El resultado que se obtiene es:

```
Datos iniciales [5, 2, 10, 9, 0, 8, 1, 5, 5, 10, 6, 4,
                 4, 2, 9, 1, 10, 10, 10]
Puntos por jugada [7, 19, 9, 9, 20, 20, 14, 6, 20, 30]
Puntaje final 154
```

6.5. Métodos disponibles para vectores

Para el manejo de vectores existe la clase *Arrays* que tiene los siguientes métodos.

Metodo	Descripción
Arrays.binarySearch(a, key)	Busca el valor *key* en un vector *a* devuelve la posición donde se encuentra o -1 si no existe.
Arrays.copyOf(original, longitud)	Copia el vector *original* en función a la longitud a otro vector
Arrays.equals(a, a2)	Devuelve verdadero si los dos vectores son iguales o falso
Arrays.fill(a, val)	Llena el vector *a* con el *val* especificado
Arrays.sort(a)	Ordena el vector *a* en forma ascendente
Arrays.toString(a)	Convierte el vector a una cadena

6.6. Ejercicios

1. Espacio de Disco
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   La eficiencia de la última computadora que compró ha estado funcionando muy rara. Usted supone que tiene mucho espacio libre en los disco de su maquina. Sin embargo el espacio libre esta distríbuido en muchos discos. Usted decide que el secreto para mejorar la eficiencia es consolidar todos los datos en la menor cantidad de discos.

   Dado un vector que representa el espacio utilizado de disco, un segundo vector que representa la capacidad total de cada disco, debe tratar de empaquetar los datos en la menor cantidad de discos. Imprimir la cantidad de discos que contienen datos después de la consolidación.

   La cantidad de discos está entre 1 y 50 elementos. Los valores de los discos están entre 1 y 1000.

   Por ejemplo si tenemos 3 discos con capacidad de $350, 600, 115$ y espacio ocupado $300, 525, 110$ veamos:

   - Disco 1 $350MB$ total, $300MB$ usados, $50MB$ libres
   - Disco 2 $600MB$ total, $525MB$ usados, $75MB$ libres
   - Disco 3 $115MB$ total, $110MB$ usados, $5MB$ libres

   Una forma de empaquetar los datos en el menor número de discos es primero mover $50MB$ del disco 3 al disco 1, llenando este completamente. La próxima vez mover los $60MB$ restantes del Disco 3 al Disco 2. Como quedan dos disco con datos después del proceso la respuesta es 2.

Input

En la entrada de datos hay varios casos de prueba. Cada caso de prueba comienza con una línea que contiene un número entero n que indica el número de discos.

Las siguiente línea contiene el espacio ocupado de los discos, separados por un espacio. Luego en la tercera línea vienen n valores representando las capacidades de los discos. La entrada termina cuando no hay más datos.

Output

Imprima en una línea el menor número de discos en los que se pueden empaquetar los datos.

Ejemplo de entrada	**Ejemplo de salida**
3 300 525 110 350 600 115 6 1 200 200 199 200 200 1000 200 200 200 200 200 5 750 800 850 900 950 800 850 900 950 1000 50 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 20 331 242 384 366 428 114 145 89 381 170 329 190 482 246 2 38 220 290 402 385 992 509 997 946 976 873 771 565 693 714 755 878 897 789 969 727 765 521 961 906	2 1 5 49 6

2. Golf Club
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un campo de golf se compone de 18 canchas de césped conocido como hoyos. El objetivo del jugador consiste en golpear una pelota con su palo de tal manera que vaya de un punto especificado en un extremo de la cancha a un punto especificado denominado *hoyo*, y hacerlo con el menor número de golpes. Asociado con cada hoyo (cancha) existe un número positivo denominado *par*, que es el número de golpes que se espera que un golfista competente realice para completar el hoyo.

El desempeño de un jugador en un hoyo está descrita por una frase que depende del número de golpes que tuvo en relación al *par*. Hacer un *bogey*, por ejemplo, significa que el jugador ha completado un hoyo en un golpe más que el valor nominal, y un *doble bogey* es de dos golpes sobre par. Dos golpes bajo par, por el contrario, es un eagle.

El siguiente es un diccionario completo de frases usadas en el golf para referirse al desempeño de un golfista:

triple bogey tres golpes sobre par

double bogey dos golpes sobre par

bogey un golpe sobre par

par exactamente par

birdie un golpe bajo par

eagle dos golpes bajo par

albatross tres golpes bajo par

hole in one exactamente un golpe

Los administradores del club de Golf le han contratado para poner en práctica, un sistema de puntuación que traduzca la jerga de arriba para hallar total numérico.

Input

La entrada consiste de varios casos de prueba. En la primera línea de cada caso de prueba se ingresa un vector de 18 elementos con el valor numérico del par para cada uno de los 18 hoyos.

Las siguientes 18 líneas tienen los resultados para cada hoyo expresados utilizando el vocabulario anteriormente descrito.

Output

Para cada caso de prueba escriba una línea con el número total de golpes que el jugador tuvo que hacer.

Ejemplo de entrada

```
1 1 1 1 1 1 1 1 1 5 5 5 5 5 5 5 5 5
bogey
bogey
bogey
bogey
bogey
bogey
```

```
bogey
bogey
bogey
eagle
eagle
eagle
eagle
eagle
eagle
eagle
eagle
eagle
3 2 4 2 2 1 1 1 3 2 4 4 4 2 3 1 3 2
eagle
birdie
albatross
birdie
birdie
par
hole in one
par
eagle
birdie
albatross
albatross
albatross
birdie
eagle
hole in one
eagle
birdie
```

Ejemplo de salida

```
45
18
```

3. Evaluando Promedios
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Para el diploma de bachillerato internacional (IB), a los estudiantes se les asigna una nota n con $1 \leq n \leq 7$, basado en exámenes tomados al final de la secundaria. Desafortunadamente, estos exámenes nunca están a tiempo para las admisiones en las universidades.

Para abordar este problema los profesores del IB tienen la necesidad de pronosticar las notas que obtendrán los estudiantes en sus exámenes. Estas predicciones tienen un impacto muy fuerte en el futuro de los estudiantes en sus posibilidades de ingresar a la universidad.

Se quiere que realice un programa para evaluar la eficiencia de las predicciones.

Por ejemplo:

Alumno	Predicción	Nota	Diferencia
1	1	3	2
2	5	5	0
3	7	4	3
4	3	5	2

Para hacer esta evaluación se imprimirá 7 porcentajes como números enteros que representan el porcentaje de notas que tuvieron diferencia de 0, de 1, de 2, sucesivamente hasta 6. En el ejemplo la respuesta es 25 0 50 25 0 0 0

Input

La entrada consiste de múltiples casos de prueba. La primera línea de cada caso de prueba contiene el número de notas n con $1 \leq n \leq 50$. La segunda línea contendrá las n notas pronosticadas separadas por un espacio. La tercera línea tendrá las n notas reales. La entrada termina cuando no hay mas datos en la entrada.

Output

La salida son 7 números enteros en una sola línea con los porcentajes mencionados.

Ejemplo de entrada

```
4
1 5 7 3
3 5 4 5
3
1 1 1
5 6 7
1
3
3
23
1 5 3 5 6 4 2 5 7 6 5 2 3 4 1 4 6 5 4 7 6 6 1
5 1 3 2 6 4 1 7 5 2 7 4 2 6 5 7 3 1 4 6 3 1 7
```

Ejemplo de salida

```
25 0 50 25 0 0 0
0 0 0 0 33 33 33
```

```
100 0 0 0 0 0 0
17 13 21 17 21 4 4
```

4. Acordes Mayores y Menores
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   En este problema su objetivo es convertir una descripción de como tocar un acorde de guitarra por su nombre. Para este problema solo consideraremos acordes mayores y menores.

   Las notas musicales tienen los siguientes 12 nombre en orden ascendente:

   $$C, C\#, D, D\#, E, F, F\#, G, G\#, A, A\#, B$$

   Las diferencia entre dos notas se llama *medio-paso*. El orden de las notas es cíclico. Esto significa que una nota una paso más arriba de B se vuelve otra vez a C. Una nota dos pasos más abajo de C es $A\#$. Todas las notas que son múltiplos de 12 medios pasos de una nota, recibe el mismo nombre. Y para nuestro propósito se consideran equivalentes.

   En este problema se considera una guitarra con 6 cuerdas, con afinación común. Las 6 cuerdas de la guitarra están afinadas en el siguiente orden: E, A, D, G, B, E.

   Si toca una nota sin presionar la guitarra en ningún traste escuchará la nota correspondiente. Si usted toca la cuerda A escuchará la nota A.

   Para cambiar la nota que uno toca puede presionar en uno de los trastes. Si usted toca una cuerda presionando el traste k usted escuchara la nota que está k medios pasos más arriba. Por ejemplo si usted toca la nota A presionando el traste 4 usted escuchara la nota $C\#$.

   Le darán un acorde de guitarra como un vector de 6 elementos, cada elemento describe una de las cuerdas en el orden explicado. Para cada cuerda nos dan el número de traste en la que se presiona la cuerda. Si no se presiona la cuerda el número 0.

   Por ejemplo el $acorde = -1, 3, 2, 0, 1, 0$. cuando toque este acorde escuchara las siguientes notas $nada, C, E, G, C, E$.

   El acorde anterior contiene tres notas distintas C, E y G. Este acorde de llama *C Mayor*

   Para cada nota X el acorde *X Mayor* se forma por tres notas distintas. Se obtiene del acorde *C Mayor* desplazando estas notas un número de medios pasos hasta que C se se convierta en C.

   Por ejemplo, si desplazamos las notas (C, E, G) tres pasos, obtenemos $(D\#, G, A\#)$ que representa el acorde *D# Mayor*.

   Similarmente los acordes de *C Menor* se forma de las notas $C, D\#$ y G, todas las notas menores son un desplazamiento de éstas.

   Dado un acorde debe decidir si es un acorde de los 12 menores o mayores como se definió. Si no es debe imprimir una línea en blanco.

   Cuando se toca una nota no interesa la cantidad de veces que se toca, solo interesa que estén las notas que se requieren.

   Para un acorde solo se tocan 3 notas.

   Si el acorde es $-1, -1, 2, 0, 1, 0$ las notas que se tocan son $nada, nada, E, G, C, E$. las notas que se tocan son C, E, G) y otra vez tenemos $CMajor$. Aunque en la teoría musical se escribe con un nombre más preciso C/E, esto es irrelevante.

Input

La entrada consiste en varios casos de prueba, Cada caso de prueba consiste en un acorde. Cada acorde está representado por 6 enteros entre 1 y 12 inclusive que representa el traste de la guitarra que presionó.

Output

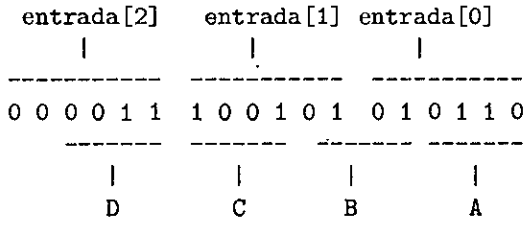
La salida es una línea por caso de prueba que debe decir "$XMayor$" o "$CMenor$" donde X es la nota. Si no es un acorde debe imprimir "".

Ejemplo de entrada	**Ejemplo de salida**
-1 3 2 0 1 0	"C Mayor"
3 2 0 0 0 3	"G Mayor"
-1 0 2 2 1 0	"A Menor"
-1 4 3 1 2 1	"C# Mayor"
8 10 10 9 8 8	"C Mayor"
0 0 0 0 0 0	""
-1 -1 4 -1 -1 7	""
-1 -1 2 0 1 0	"C Mayor"

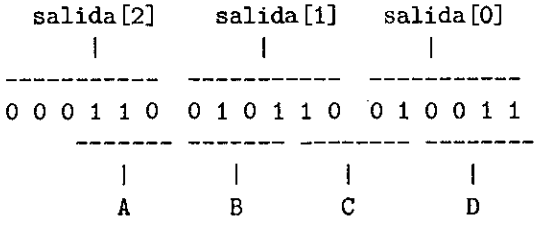
5. Transmitiendo datos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted está escribiendo una función para que dos equipos de red de diferentes vendedores se interconecten. Ellos envían y reciben datos con los mismos campos de datos, pero ordenados en forma opuesta. Dado un paquete de datos coloque en orden reverso los campos del paquete para que el otro equipo lo pueda leer.

Considere por ejemplo el paquete $n = 22, 37, 3$, con palabras de *6 bits* y cuatro campos de *4 bits*. En binario el $22 = 10110$, $37 = 100101$, $3 = 11$. Los campos se decodifican como sigue:



Donde A es el primer campo, B es el segundo campo, C es el tercero, y D el cuarto. En este ejemplo los valores de A, B, C, D son $6, 5, 9, 3$ respectivamente.La salida correcta se obtiene invirtiendo el orden D, C, B, A a A, B, C, D dando:



La salida correcta es $19, 22, 6$ que corresponden a $salida[2], salida[1].salida[0]$

Consideremos la entrada $N = 1, 0$ con longitud de palabra de 31 bits, esto significa un paquete de 62 bits, 61 bits en cero el primero en 1. Si decimos que hay 10 campos de longitud de 1 bit, una vez invertidos tendremos un 1 seguido de 9 ceros, que se convertirá en $512, 0$

Input

El paquete de entrada estará empaquetado en N palabras de longitud b bits. Estas palabras le serán dadas en un vector. El bit 0 del vector en la posición 0 es bit cero del paquete de datos, y el bit $b - 1$ del vector de entrada en la posición $N - 1$ es el bit $N * b - 1$ del paquete de datos.

La entrada consiste de varios casos de prueba. La primera línea de un caso de prueba tiene el número de elementos del vector de entrada, $(1 \leq N \leq 10)$. La segunda línea tiene los elementos del vector separados por un espacio, $(0 \leq entrada_i \leq 2^b - 1)$. La tercera línea tiene la longitud de la palabra,$(0 \leq b \leq 31)$, el número de campos,$(0 \leq nc \leq 31)$ y la longitud del campo, $(0 \leq lc \leq 31)$.

La entrada termina cuando no hay más datos.

Output

En la salida escriba las palabras del paquete, después del proceso explicado anteriormente, separadas por un espacio.

Ejemplo de entrada	**Ejemplo de salida**
3 22 37 3 6 4 4 2 1 0 31 10 1 4 1 0 0 0 10 31 1 3 15834 2483 19423 16 8 6	19 22 6 512 0 0 0 0 1 53074 60455 27516

6. Panagram
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Panagram es un programa de televisión que ofrece premios muy grandes en dinero, para detectar si una oración es un *Panagram*. Definimos un *Panagram* como una oración que contiene al menos una vez cada una de las 26 letras del alfabeto inglés.

Ejemplos:

- *the quick brown fox jumps over a lazy dog*
- *jackdawf loves my big quartz sphinx*

Cada concursante debe indicar si una oración es un *Panagram*

Input

La entrada consiste de varios casos de prueba cada uno en una línea. Cada línea contiene una oración a lo máximo 200 caracteres. Cada cadena representa una oración. Las palabras están separadas por espacios. Solo aparecerán caracteres en minúsculas.

El último caso de prueba es un asterisco.

Output

Por cada caso de prueba debe imprimir una sola línea de código que contenga *Y* si la oración es un *panagram* o *N* si no lo es.

Ejemplo de entrada

```
jackdawf loves my big quartz sphinx
abcdefghijklmnopqrstuvwxyz
hello world
*
```

Ejemplo de salida

```
Y
Y
N
```

7. Nombres de Directorio
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Su directorio contiene una lista de archivos. Todos los nombre de archivos tienen la misma longitud. Cuando escribe un comando en la forma *dir patrón*, le mostrará los nombres de archivos que igualan con el *patrón*. Un patrón solo puede contener letras de la $a - z$ el *punto* y el carácter *?*.

Cada carácter *?* empareja un solo carácter, incluyendo el *punto*. Los otros caracteres solo emparejan con otro igual.

Por ejemplo el patrón *conte?t.info* empareja con *contest.info* y *content.info*, pero no empareja con *contemnt.info* o *contests.info*.

Dados varias cadenas donde cada una representa el nombre de archivo en su directorio raíz. Escriba el patrón que empareje todos los nombres de archivos utilizando la menor cantidad de símbolos *?*.

Por ejemplo si la entrada contiene *contest.txt* y *context.txt* debe imprimir la cadena *conte?t.txt*.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba comienza en la primera línea con la cantidad de nombres de archivos a trabajar. La segunda línea tiene los nombres separados por un espacio en una sola línea. El tamaño de los nombres no excede a los 50 caracteres, y no habrán más de 50 nombres.

La entrada termina cuando no hay más datos de prueba.

Output

Para cada caso de prueba escriba una línea con el patrón correspondiente que empareja a todos los nombres de archivos.

Ejemplo de entrada

```
2
contest.txt context.txt
3
config.sys config.inf configures
3
c.user.mike.programs c.user.nike.programs c.user.rice.programs
4
a a b b
1
onlyonefile
```

Ejemplo de salida

```
conte?t.txt
config????
```

```
c.user.?i?e.programs
?
onlyonefile
```

8. Compara Cadena
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted quiere construir un programa para comparar cadenas. Usted no quiere utilizar el método normalmente disponible en el lenguaje de programación. Su programa es un nuevo método que considera dos cadenas equivalentes si difieren en un solo carácter y son del mismo tamaño,

Dadas dos cadenas debe escribir *Yes* si son equivalentes *No* en otro caso.

Por ejemplo si tenemos las cadenas a, abc, abb, bcd y ab, abc, ade, bfd La primera cadena a no es igual a ab porque difieren en el tamaño. La cadena abc es igual a la cadena abc. La tercera cadena abb no es equivalente a ade porque difiere en 2 caracteres. La última cadena bcd es equivalente a la cadena bfd porque solo difiere en el segundo carácter. La línea de respuesta para estos datos es NO, YES, NO, YES.

Input

La entrada consiste en varios casos de pruebas. La primera línea de cada caso de prueba contiene el número de cadenas a comparar. Las siguientes dos líneas contienen las cadenas. Cada línea de cadenas tendrá entre 1 y 50 elementos. Las dos líneas tienen la misma cantidad de elementos.

La entrada termina cuando no hay más datos.

Output

Por cada caso de entrada debe imprimir una línea indicando con las palabras YES o NO por cada cadena que compara. Separados por un espacio.

Ejemplo de entrada	**Ejemplo de salida**
4 a abc abb bcd ab abc ade bfd 1 adcbdeeaeafaklkfajlkfka adcbdeearafaklkfajlqfka 4 y abc bde ahsdjka y qbp fde ahsdjka	No Yes No Yes No Yes No Yes Yes

9. Directory Path
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En un sistema típico de archivos, hay muchos archivos que representan unidades completas de datos. Estos archivos se encuentran en directorios, y estos pueden estar contenidos en otros directorios, y así sucesivamente. Un camino es un puntero a un archivo o directorio específico. Los sistema operativos similares al Unix tienen un solo directorio raíz que tiene todos los otros directorios y archivos, ya sea directamente o indirectamente por medio de otros directorios. Tales sistemas operativos utilizan la estructura siguiente para los caminos.

```
/<directory-name>/<directory-name>/.../<directory-name>/<file-name>
```

Por ejemplo el archivo */etc/passwd* apunta a un archivo llamado *passwd* que está en el interior de un directorio *etc*. En este problema solo utilizaremos letras minúsculas de $a - z$.

Un caso especial es el directorio raíz al que nos referiremos con /.

Cuando un usuario está trabajando con el sistema operativo uno de los directorios se escoge como corriente. Esto permite referirse a los archivos en el directorio sin especificar el camino completo al archivo. Por ejemplo si el archivo es */home/user/pictures/me* uno puede referirse al archivo solo escribiendo *me*. Además los archivos en sub directorios también pueden referirse de una manera abreviada, por ejemplo */home/user/pictures/others/she* se puede referir como *others/she*.

Es mucho más emocionante tener referencias cortas hacia archivos que están fuera de la carpeta actual. Específicamente se utiliza .. para referirse a un directorio de un nivel superior y ../.. representa dos niveles, y así sucesivamente. Por ejemplo si usted esta en el directorio */home/user/pictures*" y quiere acceder al archivo */home/user/movies/title* con una referencia corta debe usar *../movies/title* para acceder a *title* en lugar de *../../movies/title*.

Es imposible que existan dos archivos o directorios con el mismo nombre dentro del mismo directorio y se garantiza que en los datos no existen éstos errores.

Input

La entrada consiste en varios casos de prueba, cada uno en una línea. Cada caso de prueba contiene dos cadenas separadas por un espacio. La primera es el archivo al que queremos acceder y la segunda el directorio actual. Ningún nombre de archivo termina con /. La entrada termina cuando no hay más datos.

Output

Por cada caso de entrada imprima en una línea la referencia corta para acceder al archivo o directorio.

Ejemplo de entrada

```
/home/top/data/file /home/user/pictures
/home/user/movies/title /home/user/pictures
/file /
/a/b/a/b/a/b /a/b/a/a/b/a/b
```

```
/root/root/root /root
```

Ejemplo de salida

```
../../top/data/file
../movies/title
file
../../../../b/a/b
root/root
```

10. Filtro
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted está realizando un preproceso a una consulta que toma mucho tiempo. Por esta razón cada vez que se realiza una consulta su programa debe realizar primero lo siguiente:

- Quitar todas las palabras frecuentemente utilizadas. Estas palabras no mejoran la calidad.
- Quitar todas las palabras duplicadas. Cada palabra diferente debe aparecer una sola vez.
- Ordenar las palabra restantes en orden alfabético.

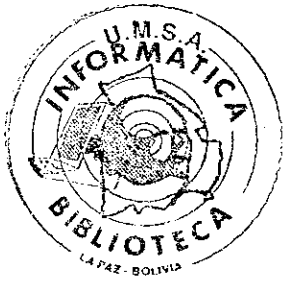
Por ejemplo si la consulta es *an easy test* y las palabras frecuentes son *a an the* la respuesta debe ser *easy, test.* Se quitó la palabra frecuente *an.*

Input

La entrada consiste en múltiples casos de prueba. Cada caso de prueba consta de dos líneas de texto. La primera línea con la consulta, y la segunda con las palabras frecuentes. La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba debe imprimir en una línea la consulta optimizada de acuerdo a la explicación anterior.



Ejemplo de entrada

```
an easy test
a an the
money money money
a an the
some really cool stuff that i forgot where to look
i the to a an that
aaaaaaaaaaaaaaaa
a
```

Ejemplo de salida

```
easy test
money
cool forgot look really some stuff where
aaaaaaaaaaaaaaaa
```

11. Evaluar Respuestas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Su profesor le ha propuesto el siguiente método simple para evaluar la correctitud de un exámen. El profesor escribe diferentes palabras claves y espera encontrar la respuesta correcta. Cada palabra tiene un puntaje asignado. Luego el profesor busca la palabra clave en las respuestas del estudiante. Si la encuentra suma el puntaje de la palabra clave a la nota del estudiante. Si encuentra la palabra clave múltiples veces solo suma una sola vez.

Veamos un ejemplo, Supongamos que las palabras claves de profesor son: $red, fox, lazy, dogs$ que tienen el puntaje $25, 25, 25, 25$ respectivamente. Si la frase a evaluar es *the quick brown fox jumped over the lazy dog*, podemos encontrar fox y dog por esto su nota es 50.

Input

La entrada consiste en varios casos de prueba. cada caso de prueba viene en cuatro líneas, la primera línea tiene el número de palabras claves, la segunda línea las palabras claves. La tercera línea los puntajes, uno por palabra clave. La cuarta línea es la frase que hay que evaluar. Las palabras claves son entre 1 y 25 inclusive. Cada palabra tendrá entre 1 y 50 caracteres, inclusive. La frase de respuesta tendrá entre 1 y 50 caracteres inclusive. La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba escriba una línea con el puntaje que le corresponde.

Ejemplo de entrada

```
4
red fox lazy dogs
25 25 25 25
the quick brown fox jumped over the lazy dog
4
red fox lazy dogs
25 25 25 25
highschool matches are nice
4
red fox lazy dogs
1 2 3 4
lazy lazy lazy lazy
```

Ejemplo de salida

```
50
0
3
```

12. Decodificando Palíndromes
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A usted le dan una cadena código, una lista de *posición* y una lista de *longitud*. El código contiene una cadena codificada que se decodifica con el método siguiente: Recorra todos le elementos desde $posicion[i]$, en orden. Para cada elemento i, tome la subcadena de longitud $longitud[i]$, e inserte la subcadena en orden inverso antes de la $posicion[i] + longitud[i]$ creando una subcadena palíndrome.

Por ejemplo tomemos la cadena *Misip* y el vector de posición $2, 3, 1, 7$ y longitud $1, 1, 2, 2$. Primero tomamos la subcadena de la posición 2 y longitud 1 que es la s e insertamos la subcadena para obtener *Missip*. Segundo tomamos la cadena de la posición 3 y de longitud 1 que es s y *Misssip*. Luego tomamos la tercera posición y longitud 1 y 2 respectivamente, esta subcadena en orden inverso es *si*. La posición 1 más la longitud 2 da la posición 3, insertamos en ésta posición la subcadena obtenida *si* para tener *Mississip* continuamos con el último caso, que es, posición 7 longitud 2 dando la cadena inversa *pi* que se inserta en la posición 9 para obtener *Mississippi*.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba contiene 4 líneas. La primera línea consiste en la cadena código, que es entre 1 y 10 caracteres. La segunda línea contiene un número entero con la cantidad de números de posición que existen, este es un número entre 0 y 10. La tercera línea tiene todas las posiciones que son números enteros separados por un espacio. La cuarta línea tiene todas las longitudes que son la misma cantidad que las posiciones, son números enteros separados por un espacio.

La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba imprima una línea con la cadena obtenida del proceso anterior.

Ejemplo de entrada	Ejemplo de salida
ab 1 0 2 Misip 4 2 3 1 7 1 1 2 2 XY 4 0 0 0 0 2 4 8 16 TC206 3 1 2 5 1 1 1	abba Mississippi XYYXXYYXXYYXXYYXXYYXXYYXXYYXXYYX TCCC2006

13. Voto del Conejo
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los conejos muchas veces están tristes, así que un grupo de conejos decidió realizar un concurso de belleza para determinar cual tiene las orejas más bellas.

Las reglas para el concurso son como sigue: Cada conejo envía un voto. Si un conejo vota por si mismo el voto es inválido y no se cuenta. Al final el conejo que tenga más votos gana. En caso de empate no hay ganador.

Por ejemplo si los votos son *Alice*, *Bill*, *Carol*, *Dick*, *Bill*, *Dick*, *Alice*, *Alice* significa que *Alice* voto por *Bill* por *Dick*, *Carol* por *Alice*, y *Dick* por *Alice*. Por lo que *Alice* obtuvo la mayoría de los votos.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba tiene tres líneas. La primera línea contiene el número ($1 \leq N \leq 50$)de conejos que votaron. La segunda línea contiene los nombres de los conejos que votaron, separados por un espacio. La tercera línea contiene los votos que hizo cada uno como se explicó. La entrada termina cuando no hay más datos.

Output

En la salida imprima por cada caso de prueba el nombre del conejo ganador. En caso de empate imprima una línea en blanco.

Ejemplo de entrada

```
4
Alice  Bill  Carol  Dick
Bill  Dick  Alice  Alice

4
Alice  Bill  Carol  Dick
Carol  Carol  Bill  Bill

4
Alice  Bill  Carol  Dick
Alice  Alice  Bill  Bill

2
Alice  Bill
Alice  Bill

4
WhiteRabbit  whiterabbit  whiteRabbit  Whiterabbit
whiteRabbit  whiteRabbit  whiteRabbit  WhiteRabbit
```

Ejemplo de salida

```
Alice

Bill

whiteRabbit
```

14. Sorteos Bancarios
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Muchos bancos locales han comenzado un sistema de sorteos en lugar de pagar intereses en la forma tradicional. Es más barato y mucha gente no ve la diferencia. El sistema de sorteos funciona como sigue:

Al final de cada semana se realiza un sorteo. Cada cliente que posee una cuenta recibe un cupón por cada peso en su saldo de cuenta. Después que se han distribuido todos los cupones, se escoge un cupón al azar. Cada cupón tiene la misma probabilidad de salir escogido. El poseedor del cupón gana un premio que es inmediatamente adicionado a su cuenta.

Usted acaba de abrir una cuenta en el banco y desea conocer su saldo esperado al momento, en el futuro. Usted curiosamente ha podido conocer los saldos de todos los clientes del banco. Estos balances son dados como un vector siendo el primer elemento su saldo.

Para este problema tomen en cuenta que no hay transacciones adicionales al sorteo semanal. Tampoco se crean o cierran cuentas.

Conocedor de sus conocimientos de programación le han pedido calcular su saldo esperado después de un número dado de semanas.

Veamos un ejemplo:

Supongamos que hay tres clientes cada uno con 2 pesos de saldos, es decir, usted también tiene 2 pesos. El premio asignado es de 1 peso. Cuál es su saldo probable después de la segunda semana?

Considere para la solución que la probabilidad de que los balances en la primera semana queden $3, 2, 2$ es de $1/3$. La probabilidad de que queden $2, 3, 2$ esta también de $1/3$ probabilidad y que se conviertan en $2, 2, 3$ es $1/3$. Esto nos da un saldo esperado es de 2,6666666666666665.

Input

La entrada consiste en múltiples casos de prueba que terminan cuando no hay más datos. La primera línea contiene un número entero ($1 \leq n \leq 50$) que representa el número de cuentas. La segunda línea contiene n números enteros con el saldo de las cuentas. El primer valor representa su saldo. Por lo menos un saldo es mayor a cero. La tercera línea contiene un entero entre 1 y 1000 inclusive que indica el monto del premio. La cuarta línea contiene un entero entre 1 y 1000 inclusive que indica el número de semanas

La entrada termina cuando no hay más datos.

Output

La salida consiste de una sola línea que contiene el saldo esperado. El resultado debe imprimirse con error menor a $1e-9$. Si no tiene dinero no puede ganar y la respuesta deberá ser "0.0".

Ejemplo de entrada	**Ejemplo de salida**
2 100 100 100 2 3 2 2 2 1 2 1 2 3 4 5 6 7 8 9 10 100 20 7 0 200 200 0 300 300 600 3 776	200.0 2.6666666666666665 14.0 306.0 0.0

15. Código de verificación
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A usted le dan una cadena de código que contiene un mensaje enteramente compuesto por dígitos del 0 al 9. Cada dígito consiste de un número de líneas segmentadas (guiones horizontales y guiones verticales). El código de verificación del mensaje se definirá como el número de guiones que se pueden encontrar en el mensaje.

La figura muestra como se construyen los números.

Como puede ver el dígito cero tiene 6 líneas y el 1 tiene solo dos.

Input

La entrada consiste de varios casos de prueba. La primera línea es un número entero que indica el número de casos de prueba. Cada una de las líneas siguientes es un caso de prueba, representando el mensaje a verificar.

La entrada termina cuando no hay más datos.

Output

Para cada caso de prueba imprima el código de verificación que se encuentra en el mensaje.

Ejemplo de entrada	**Ejemplo de salida**
13579	21
02468	28
732543709328750020279632950521 75	157

16. Caballerosidad

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Dos colas de personas a menudo tienen que fusionarse en una sola cola. Pero, la caballerosidad no ha muerto cuando un hombre y una mujer están a punto de entrar a una sola cola, el hombre siempre sede el lugar a la mujer.

Se tienen dos colas donde hay hombres y mujeres, escribir un programa que ordene según el género de las personas en ambas colas. Si dos mujeres se encuentran al frente de ambas colas, la mujer de la primera línea va primero. Del mismo modo, si dos hombres están en la parte delantera de ambas colas, el hombre de la primera cola va primero.

Entonces, las personas en la parte delantera de ambas líneas se comparan de nuevo. Cada cola de entrada es una cadena de letras, cada letra representa a un hombre o una mujer. Cada hombre será representada por una M y cada mujer por una W. La salida debe ser de la misma forma.

La parte izquierda de una cola representa a la cabeza de la cola.

Input

La entrada de datos consiste de dos cadenas cada una representa una cola, ambas colas tienen entre $[1 - 50]$ caracteres, los únicos caracteres que tendrán estas dos cadenas son M y W, la entrada termina cuando no haya más casos de prueba.

Output

Para cada caso de entrada mostrar la cadena que representa la cola ordenada de acuerdo a los requerimientos presentados al principio.

Ejemplo de entrada	**Ejemplo de salida**
M W MM MW MMMM W M WWW MMMMMMM W WWWWWW M	WM MMMW WMMMM WWWM WMMMMMMM WWWWWWM

17. Contenedores

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Hay varios contenedores vacíos alíneados en una fila, y se desea poner paquetes en ellos. Se empieza con el primer contenedor y el primer paquete. Haga lo siguiente hasta que todos los paquetes se encuentran dentro de los contenedores, use la siguiente información:

a) Si el paquete actual no cabe en el contenedor actual, vaya al paso 3. De lo contrario, vaya al siguiente paso.

b) Ponga el paquete actual en el contenedor actual. Tome el siguiente paquete, y volver al paso 1.

c) Dejc el contenedor actual a un lado (que no pondrá contener ningún paquete más). Pasar al siguiente contenedor y volver al paso 1.

No se permite reordenar los contenedores o los paquetes.

Input

La entrada consiste en varios casos de prueba. La primera línea de cada caso de prueba contiene el número de contenedores C. La segunda línea contiene $1 < C \leq 50$ elementos donde cada elemento representa la capacidad de un contenedor. La tercera línea contiene el número de paquetes $1 < P \leq 1000$. La cuarta línea contiene los tamaños de los P paquetes. Se garantiza que se puede colocar todos los paquetes en los contenedores. La entrada termina cuando no hay más datos.

Output

Mostrar la suma de los espacios perdidos de todos los recipientes luego de haber ubicado todos los paquetes en los recipientes.

Ejemplo de entrada	**Ejemplo de salida**
3 5 5 5 3 5 5 5 3 5 6 7 3 5 5 5 3 2 3 5 1 3 4 3 4 5 6 5 3 3 3 3 3	0 3 7 3

18. Cortando Postes

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un trabajador descuidado ha plantado varios postes en una fila para construir una cerca. Todos ellos deben tener la misma altura pero fueron cortadas en diferentes tamaños. El propietario no quiere solamente que todos estén a la misma altura también quiere que la altura sea la más alta posible. Nuestra tarea es cortar las cimas más altas de los postes y pegarlas como tapas en la parte superior de las más cortas. Para ello, primero ordenar los postes del más alto al más bajo, y proceder como sigue:

a) Cortar la punta del poste más alta, dejando a su altura igual a la altura media de los postes (por lo que el poste no se corta más).

b) Pegar esta pieza en la parte superior de las más cortas del poste.

c) Reordenar los postes, y continuar desde el primer paso hasta que todos los postes sean de la misma altura.

Escriba un programa que lea la altura de postes y devuelva el número de cortes necesarios para que todos los postes estén a la misma altura mediante el algoritmo descrito.

Input

La entrada consiste en un lote de enteros entre $[1-50]$ los mismos representan la altura de los postes de la cerca, la altura está entre $1 \leq alturaposte \leq 1000$, se garantiza que el promedio de la altura de los postes es un valor entero.

La entrada termina cuando no hay más datos.

Output

Para cada caso de entrada mostrar el número de cortes necesarios para dejar los postes de la cerca a una misma altura.

Ejemplo de entrada	**Ejemplo de salida**
2 1 3 3 10 10 10 4 1 1 3 3 8 10 10 10 10 10 10 10 18 10 10 1 9 2 8 3 7 4 6 10	1 0 2 7 5

19. Comprando

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Steve desea comprar un coche nuevo. No es rico por lo que el preferirá un coche razonablemente barato. El único inconveniente es que la calidad de los coches baratos es cuestionable. Así, Steve decide hacer una lista de precios de los automóviles de donde escogerá el coche con el tercer precio más bajo.

Se lee un vector de enteros con los precios. Un precio se puede repetir varias veces en la lista de precios, pero debe contar como una sola vez. Escribe un programa que devuelve el tercer precio más bajo de la lista. Si hay menos de tres precios diferentes el resultado es -1.

Input

La entrada consiste de varios casos de prueba. La primera línea de cada caso de prueba contiene el número de precios coches $1 < N \leq 50$. La siguiente línea contiene $0 < N \leq 1000$ números que representan la lista de precios. La entrada termina cuando no hay más datos.

Output

Imprima para cada caso de prueba el tercer precio menor de la lista.

Ejemplo de entrada

```
9
10  40  50  20  70  80  30  90  60
10
10  10  10  10  20  20  30  30  40  40
1
10
5
80  90  80  90  80
```

Ejemplo de salida

```
30
30
-1
-1
```

Capítulo 7

Arreglos multidimensionales

7.1. Definición

Para introducir los arreglos multidimensionales, definamos un vector como sigue:

```
int [] v;
```

Al colocar dos corchetes hemos indicado que se trata del arreglo de una dimensión. Si escribimos

```
int [][] v;
```

dos veces los corchetes hemos definido un arreglo de dos dimensiones. Los arreglos bidimensionales también se conoce como matrices. Definamos un arreglo bidimensional de $2x3$,

```
int [][] v= new int[2][3];
```

Este arreglo tiene 2 filas por 3 columnas. Para definir valores constantes la sintaxis es como sigue:

```
int [][] x= {{5,2,10},{9,0,8}};
```

Bien ahora el resultado de *System.out.println(x.length)* es 2 por que tiene 2 filas. El resultado de *System.out.println(x[0].length)* es de 3 por que la fila 0 tiene tres elementos.

Para acceder a un elemento específico se requieren 2 índices. El de la fila y el de la columna. Así $x[1][0] = 9$, esto es la fila 1 columna 0.

En arreglos multidimensionales no es posible utilizar *Arrays.toString(x)* para ver el arreglo como cadena. Esto se debe a que cada elemento del vector es un puntero a otro vector. Para hacer esto tenemos dos opciones: la primera recorrer con la instrucción *for*

```
for (int  i=0; i<x.length;i++)
    System.out.println(Arrays.toString(x[i]));
```

y la segunda utilizando la instrucción *for each*

```
for (int [] i: x)
    System.out.println(Arrays.toString(i));
```

Los arreglos de dos dimensiones se denominan matrices. Supongamos que queremos mostrar matriz del ejemplo anterior como sigue:

```
(0,0)=5 (0,1)=2 (0,2)=10
(1,0)=9 (1,1)=0 (1,2)=8
```

El código nos muestra el recorrido del vector y como podemos armar ésta salida.

```
public class programa2 {
    public static void Mat1(String[] args) {
        int [][] x= {{5,2,10},{9,0,8}};
        for (int i=0; i<2;i++){
            for (int j=0; j<3;j++)
                System.out.print(''('+i+','+j+')='+x[i][j]+' ');
            System.out.println();
        }
    }
}
```

Cada vez que terminamos de mostrar una fila bajamos imprimimos un cr y lf.

7.2. Ejercicios clásicos

1. Escriba un programa para llenar de ceros la matriz triangular superior de una matriz de dimensión NxN.
2. Escriba un programa para llenar de ceros la matriz triangular inferior de una matriz de dimensión NxN.
3. Escriba un programa para imprimir una matriz de dimensión NxM por filas
4. Escriba un programa para imprimir una matriz de dimensión NxM por columnas
5. Escriba un programa para sumar dos matrices de NxM Cada valor se calcula como sigue: $(c(i,j) = a(i,j) + b(i,j))$.
6. Escriba un programa para multiplicar dos matrices A, B. Dados $A = (a_{ij})_{m.n}$ $B = (b_{ij})_{n.p}$, el producto se define como $AB = (C_{ij})_{m.p}$ donde

$$c_{ij} = \sum_{r=1}^{n} a_{ir} a_{rj}$$

7. Escriba un programa para hallar la matriz transpuesta de una matriz cuadrada de dimensión MxM.
8. Escriba un programa para generar una matriz caracol de unos y ceros
9. Escriba un programa que lea un número impar del teclado y genere una matriz cuadrada donde la suma de las filas, columnas y diagonales da el mismo número. Por ejemplo en la matriz siguiente la suma es 15

6	1	8
7	5	3
2	9	4

Para ejemplificar resolvamos el ejercicio del cuadrado mágico. El truco para construir un cuadrado de dimensión impar, esto es, el número de elementos de la fila es impar, es comenzar en el medio de la primera fila.

0	1	0
0	0	0
0	0	0

Ahora avanzamos un lugar hacia arriba, en diagonal, para no salirnos del cuadrado hacemos que el movimiento sea cíclico.

0	1	0
0	0	0
0	0	2

Repetimos el movimiento

0	1	0
3	0	0
0	0	2

Ahora no podemos continuar en diagonal, porque la casilla de destino está ocupada, o sea que bajamos un fila.

0	1	0
3	0	0
4	0	2

Continuamos

0	1	6
3	5	0
4	0	2

Luego se baja un lugar

0	1	6
3	5	7
4	0	2

Y termina en:

8	1	6
3	5	7
4	9	2

El programa siguiente construye el cuadrado mágico.

```
import java.util.Scanner;
public class Cuadrado {

     public static void main(String[] args) {
        Scanner lee=new Scanner(System.in);
         int N = lee.nextInt();

         int[][] magico = new int[N][N];

         //Comenzamos en fila 0 y al medio con 1
         int fila = 0;
         int col = N/2;
         magico[fila][col] = 1;

         for (int i = 2; i <= N*N; i++) {
             // si la diagonal es 0 recorrer un lugar
             if (magico[(fila - 1+N) % N][(col + 1) % N] == 0) {
                 fila = (fila - 1+N) % N;
                 col = (col + 1) % N;
                 }
             else {
                 fila = (fila + 1) % N; //bajar una fila
                 // don't change col
                 }
             magico[fila][col] = i;
             }

         // print results
         for (int i = 0; i <N; i++) {
             for (int j = 0; j <N; j++) {
                 System.out.print(magico[i][j] + ''\t");
                 }
             System.out.println();
             }

         }
}
```

En el programa la función módulo, hace que recorramos en forma cíclica ya sea por filas o columnas.

7.3. Dimensión de tamaño variable

Hasta este momento hemos visto solo como definir arreglo donde el tamaño es fijo. Consideremos la definición siguiente:

```
int [][] v;
```

Esto especifica que v es un arreglo bidimensional, pero, no se ha especificado el tamaño. El valor inicial en éste caso es *null*. Con el siguiente código podemos especificar el tamaño del vector:

```
v=new int [m][n] ;
```

Para hacer que cada fila tenga un tamaño diferente: Primero definimos un vector sin especificar la dimensión de las columnas, solo especificamos la dimensión de las filas, digamos 5:

```
int [][] v= new int[5][];
```

Luego para cada fila especificamos su tamaño:

```
v[0]=new int [i];
```

Esto nos puede ahorrar mucho espacio de memoria. Por ejemplo para definir una matriz triangular inferior, cada fila la definiríamos como sigue:

```
for (int  i=0; i<v.length;i++)
    v[0]=new int [i];
```

7.4. Arreglos dinámicos

Los arreglos que pueden cambiar su tamaño manteniendo sus valores se denominan dinámicos. Los arreglos o vectores dinámicos son dos: *ArrayList* y *Vector*. Estas dos estructuras son similares con la diferencia que *Vector* cada vez que se incrementa de tamaño duplica su capacidad. Comienza en 10 luego duplica a 20, después a 40 y así sucesivamente. Veamos un ejemplo

```
public class VectorDinamico {

    {
        private static final String letras[] =
        { ''a", ''b", ''c", ''d",''e", ''f" };

        public static void main(String args[]) {
            Vector<String>vector = new Vector<String>();
            System.out.println(vector);
            System.out.printf(''\nTamaño: % d\nCapacidad: % d\n",
            vector.size(), vector.capacity());
            // añadir elementos
            for (String s : letras)
                vector.add(s);
            System.out.println(vector);

           System.out.printf(''\n Tamaño: %d \n Capacidad: %d \n",
            vector.size(), vector.capacity());

            // añdir mas elementos
            for (String s : letras)
                vector.add(s);
            System.out.println(vector);
            System.out.printf(''\nTamano: % d\n Capacidad: % d\ n"
            vector.size(), vector.capacity());
            }
        }
```

El resultado de la ejecución es como sigue:

```
[]

Tamaño: 0
Capacidad: 10
[a, b, c, d, e, f]

Tamaño: 6
Capacidad: 10
[a, b, c, d, e, f, a, b, c, d, e, f]

Tamaño: 12
Capacidad: 20
```

Antes de insertar valores el vector está vacío. Se insertan 6 valores y tiene una capacidad de 10. Cuando se inserta 6 valores adicionales el tamaño crece a 12 pero la capacidad se ha duplicado a 20.

Los métodos de Java para la colección *Vector* se describen en el cuadro

Método	**Descripción**
add(E elemento)	Agrega un elemento al vector.
clear()	Borra la lista.
capacity()	Muestra el espacio reservado para el vector.
addAll(int i,coleccion)	Agrega toda una colección.
addAll(int i,coleccion)	Agrega toda una colección en la posición i.
remove(int i)	Quita el elemento de la posición i.
set(int i, E elemento)	Cambia el elemento de la posición i.
isEmpty()	Devuelve true si el vector esta vacío.
contains(E elemento)	Devuelve true si el vector contiene el elemento.
size()	Devuelve el número de elementos.
firstElement()	Devuelve el primer elemento.
lastElement()	Devuelve el último elemento.

Los *ArrayList* administran la memoria como una lista y crecen dinámicamente. La capacidad es igual al tamaño. Veamos un ejemplo de como puede crecer un *ArrayList* de tamaño.

```
import java.util.ArrayList;
public class Crece {
     public static void main(String args[]) {
         ArrayList v = new ArrayList(5);
         for (int i = 0; i <10; i++) {
             v.add(i);
             }
         System.out.println(v);
         }
}
```

En este ejemplo hemos definido una vector del tipo *ArrayList* con tamaño 5. Porteriormente se insertan 10 valores. Cuando se llega al límite de elementos el vector cambia dinámicamente de tamaño incrementando el mismo en uno. Cabe notar que no es posible acceder a valores que exceden el tamaño del mismo.

Los métodos de Java para la colección *ArrayList* se describen en el cuadro

Método	**Descripción**
add(E elemento)	Agrega un elemento al vector.
clear()	Borra la lista.
addAll(int i,coleccion)	Agrega toda una colección.
addAll(int i,coleccion)	Agrega toda una colección en la posición i.
remove(int i)	Quita el elemento de la posición i.
set(int i, E elemento)	Cambia el elemento de la posición i.
isEmpty()	Devuelve true si el vector está vacío.
contains(E elemento)	Devuelve true si el vector contiene el elemento.
size()	Devuelve el número de elementos.

Como ejemplo vamos a crear una lista con 5 números enteros, hacer crecer la lista, modificar y listar los elementos de la misma.

```
import java.util.ArrayList;
public class EjemploArrayList {
     public static void main(String args[]) {
         ArrayList v = new ArrayList(5);
         for (int i = 0; i <10; i++) {
             v.add(i);
             }
         // imprimir los elementos
         System.out.println(v);
         // imprimir el elemento 5
         System.out.println(v.get(5));
         // cambiar el elemento 3 por 100
         v.set(3, 100);
         // Eliminar el elemento 5
         v.remove(5);
         // imprimir el tamaño
         System.out.println(v.size());
         // recorrer la lista
         for (int i = 0; i <v.size(); i++) {
             System.out.print(''Elemento " + i
             + ''=" + v.get(i) + '' ");
             }

         }
}
```

7.5. Arreglos de más de dos dimensiones

Para definir arreglos de cualquier dimensión se ponen más corchetes en c la definición. Por ejemplo para definir un arreglo de tres dimensiones:

```
int [][][] v;
```

Para acceder a los valores del arreglo requeriremos tres índices. Por ejemplo para definir un cubo con caras de tres por tres, hay que especificar 6 caras y el tamaño de cada cara:

```
int [6][3][3] v;
```

7.6. Ejemplo de aplicación

Resolvamos el ejemplo Buscando Pares de letras cuyo enunciado dice:

Juan ha decido estudiar las frecuencias de aparición de dos letras en un texto. Para esto dispone una cantidad de texto. Para entender el problema realicemos un ejemplo corto . Su pongamos que el texto es casa cosa amor roma saca en este texto podemos los siguientes pares de letras *ca*, *as*, *sa*, *co*, *os*, *sa*, *am*, *mo*, *or*, *ro*, *om*, *ma*, *sa*, *ac*, *ca*. Analizando las frecuencias de cada par de letras es: $ca = 2$, sa = 3 el resto tiene una sola aparición.

Input

Hay varios casos de prueba, cada uno en una línea. Cada línea es una cadena de longitud arbitraria. Y termina cuando no hay más datos en la entrada.

Output

La salida consiste del par de letras y la frecuencia separados por un espacio. Solo se listarán los pares de letras que tengan una frecuencia mayor a 3. La salida estará ordenada en orden alfabético, primero por la primera letra luego por la segunda. Después de cada caso de prueba imprima ----------.

El ejemplo de datos de entrada es:

```
casa cosa amor roma saca aca
cocacola cocacola todoas a la cola
coco coco coco coco toma cocacola
```

El ejemplo de datos de salida es:

```
----------
co 5
la 4
----------
co 10
oc 5
----------
```

Para contar las combinaciones de letras primero recordamos que cada carácter es un número entre 0 y 255. Inicialmente leemos una línea con $texto = en.nextLine().trim()$. Usamos el método $trim()$ para eliminar los espacios al final de la línea.

Para contar los pares de caracteres definimos una matriz de $255x255$. La elección del tamaño es debido a que existen 255 caracteres. Para contar los pares de caracteres los usamos como índice. El código $matriz[texto.charAt(i)][texto.charAt(i+1)]++$ incrementa en uno la ocurrencia de cada par de caracteres.

Cuando recorremos la matriz el índice representa el caracter por esto cuando se imprimen colocamos $(char)i$. EL programa resultante es:

```
import java.util.Scanner;
public class BuscaPar {

    public static void main(String[] args) {
        Scanner en =new Scanner(System.in);
        int [][] matriz = new int [255][255];
        String texto="";
        while (en.hasNextLine()){
            texto=en.nextLine().trim();
            matriz = new int [255][255];
            for (int i=0;i<texto.length()-1;i++)
                if(texto.charAt(i)==" || texto.charAt(i+1)==")
                continue;
            else
                matriz[texto.charAt(i)][texto.charAt(i+1)]++;
            for (int i=0;i<255;i++)
                for (int j=0;j<255;j++)
                if(matriz[i][j]>3)
                System.out.println(""+(char)i+(char)j+" "+matriz[i][j]);

            System.out.println("----------");
        }

    }
}
```

7.7. Ejercicios

1. Regando el jardín
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Nuestro jardín es un cuadrado que contiene n filas y n columnas. y contiene $n * n$ plantas. Se tiene la distancia f entre filas de plantas y la distancia c entre columnas.

   Quiero regar el jardín sin embarrarme los zapatos. Esto requiere que me pare fuera del jardín. La manguera tiene un alcance d desde la posición donde me paro. Por supuesto me puedo mover al contorno del jardín.

   Se trata de conocer el número de plantas que no pueden ser regadas. Por ejemplo: si $n = 3$, tenemos un total de 9 plantas, $f = 2$ indica que la distancia en filas de plantas es 2, $c = 1$ indica que la distancia entre columnas es de una columna. Esto no da la siguiente representación:

```
    0  1  2  3  4  5  6  7  8
0   ooooooooooooooooooooooooo
    o  |  |  |  |  |  |  |  o
1   o--+--P--+--P--+--P--+--o
    o  |  |  |  |  |  |  |  o
2   o--+--P--+--P--+--P--+--o
    o  |  |  |  |  |  |  |  o
3   o--+--P--+--P--+--P--+--o
    o  |  |  |  |  |  |  |  o
4   ooooooooooooooooooooooooo
```

   En esta representación las letras P representan la posición donde se colocan las plantas. Si la manguera puede regar una distancia $d = 2$, desde la posición para regar que esta marcada con o podemos regar todas las plantas por lo que quedan 0 plantas sin regar.

Input

La entrada consiste de varios datos de prueba, uno por línea. En cada línea están los datos n, f, c, d separados por un espacio, con $1 \leq n, f, c \leq 50$ y $1 \leq d \leq 10,000$. La entrada termina cuando no hay más datos.

Output

En la salida debe escribir una línea por caso de prueba con el número de plantas que no podrán ser regadas.

Ejemplo de entrada	**Ejemplo de salida**
3 2 1 2	0
3 2 1 1	3
6 2 5 5	8
6 2 5 3	24
50 50 50 49	2500

2. Cuadrados de números
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Considere la siguiente matriz de números

1	0	3	4	1
4	5	8	15	20
1	10	23	46	81
0	11	44	113	240
3	14	69	226	579

Comenzando en la columna superior y la fila de la izquierda cada número es igual a la suma de los números inmediatamente a la izquierda, arriba, y arriba a la izquierda. Dado un vector que representa la primera fila y la primera columna debe hallar el valor de la última fila y última columna.

Input

La entrada consiste de varios casos de prueba, En la primera línea del caso de prueba está el número de elementos de las filas, que es el mismo de las columnas $2 \leq n \leq 10$, el primer elemento de las primera fila es el mismo que el de la primera columna. Los elementos de la primera fila y columna son números entre 0 y 9 respectivamente. La entrada termina cuando no hay más datos de prueba.

Output

Por cada caso de entrada imprima en una línea el contenido de la casilla de la última fila y columna, después de calcular éste de acuerdo a lo descrito.

Ejemplo de entrada	Ejemplo de salida
5 1 0 3 4 1 1 4 1 0 3 10 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	579 13163067 0

3. Buscaminas

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

¿Quién no ha jugado al Buscaminas? Este entretenido juego acompaña a cierto sistema operativo cuyo nombre no logramos recordar. El objetivo del juego es encontrar todas las minas ubicadas en un campo de dimensiones $M \times N$.

El juego muestra un número en un recuadro que indica la cantidad de minas adyacentes a ese recuadro. Cada recuadro tiene, como mucho, ocho recuadros adyacentes, arriba, abajo, izquierda, derecha y diagonales.

El campo de ejemplo , de tamaño 4×4 que se muestra a la izquierda contiene dos minas, cada una de ellas representada por el carácter $*$. Si representamos el mismo campo con los números descritos anteriormente, tendremos el campo de la derecha:

```
*...        *100
....        2210
.*..        1*10
....        1110
```

Input

La entrada consta de un número arbitrario de campos. La primera línea de cada campo consta de dos números enteros, n y m ($0 \leq n, m \leq 100$), que representan, respectivamente, el número de líneas y columnas del campo. Cada una de las siguientes n líneas contiene, exactamente, m caracteres, que describen el campo.

Los recuadros seguros están representados por "." y los recuadros con minas por "$*$", en ambos casos sin las comillas. La primera línea descriptiva de un campo en la que $n = m =$ representa el final de la entrada y no debe procesarse.

Output

Para cada campo, escribir el mensaje *Field #x:* en una línea, donde x corresponde al número del campo, empezando a contar desde 1. Las siguientes n líneas deben contener el campo con los caracteres "." sustituídos por el número de minas adyacentes a ese recuadro. Debe haber una línea en blanco entre los distintos campos mostrados.

Ejemplo de entrada	**Ejemplo de salida**
4 4 *... .... .*.. .... 3 5 **... ..... .*... 0 0	Field #1: *100 2210 1*10 1110 Field #2: **100 33200 1*100

4. Bubble Mapas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Bubble Inc Esta desarrollando una nueva tecnología para recorrer un mapa en diferentes niveles de zoom. Su tecnología asume que la región m se mapea a una región rectangular plana y se divide en sub-regiones rectangulares que representan los niveles de zoom.

La tecnología de Bubble Inc. representa mapas utilizando la estructura conocida como quad-tree.

En un quad-tree, una región rectangular llamada x puede ser dividida a la mitad, tanto horizontalmente como verticalmente resultando en cuatro sub regiones del mismo tamaño. Estas sub regiones se denominan regiones hijo de x, y se llaman xp para la superior izquierda, xq para la superior derecha, xr de abajo y a la derecha y xs para las de abajo a la izquierda xc representa la concatenación de la cadena x y carácter $c = p, q, r$ o s. Por ejemplo si la región base mapeada se denomina m, las regiones hijo de m son de arriba y en el sentido del reloj: mp, mq, mr y ms como se ilustra.

mp	mq
ms	mr

Cada sub región puede ser subdividida. Por ejemplo la región denominada ms puede ser sub dividida en más sub regiones msp, msq, msr y mss, como se ilustra.

msp	msq
mss	msr

Como otro ejemplo la figura muestra el resultada de dividir las sub regiones hijo de llamada msr:

msrpp	msrpq	msrqp	msrqq
msrps	msrpr	msrqs	msrqr
msrsp	msrsq	msrrp	msrrq
msrss	msrsr	msrrs	msrrr

Los nombres de la sub regiones tienen la misma longitud del nivel de zoom, dado que representan regiones del mismo tamaño. Las sub regiones en el mismo nivel de zoom que comparten un lado común se dicen vecinos. Todo lo que está fuera de la región base m no se mapea para cada nivel de zoom todas las sub regiones de m son mapeadas.

La tecnología de mapas Bubble's le provee al usuario una forma de navegar de una sub región a una sub región vecina en las direcciones arriba, abajo, izquierda y derecha. Su misión es la de ayudar a Bubble Inc. a encontrar la sub región vecina de una sub región dada. Esto es que dado el nombre de una sub región rectangular usted debe determinar los nombres de sus cuatro vecinos

Ejercicios adicionales sobre el mismo problema

- Al subir en uno el nivel del zoom, determinar a que región pertenece.
- Determinar número de niveles de zoom con los que se llega a la región.
- Escribir las regiones que son diagonales a la región dada.

Input

La entrada contiene varios casos de prueba. La primera línea contiene un entero representando el número de casos de prueba. La primera línea contiene un entero N indicando el número de casos de prueba. Cada una de las siguientes N líneas representan un caso de prueba conteniendo el nombre la región compuesta por C caracteres ($2 \leq C \leq 5000$), la primera letra siempre es una letra m seguida por una de las siguientes p, q, r o s.

Output

Para cada caso de prueba su programa debe producir una línea de salida, que contiene los nombres de las cuatro regiones de una región dada, en el orden de arriba abajo izquierda y derecha. Para vecinos que no está en mapa debe escribir *none* en lugar de su nombre. Deje una línea en blanco entre dos nombres consecutivos.

Ejemplo de entrada	**Ejemplo de salida**
2 mrspr mps	mrspq mrssq mrsps mrsqs mpp msp <none> mpr

5. Cuadros Estadísticos
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

La oficina de estadísticas de su comunidad ha realizado muchas encuestas y requiere que realice una serie de cuadros para la publicación de resultados. Como hay mucho trabajo le ha encargado realizar un programa que permita obtener totales, promedios y porcentajes sobre los resultados.

Input

La entrada consiste de varios cuadros. La primera línea tiene un número entero que indica la cantidad de cuadros que vienen a continuación. La siguiente línea contiene dos números n, m con $2 \leq m, n \leq 1000$, que representan la filas y columnas del cuadro (tabla). Las siguientes líneas contienen los valores de la matriz separados por espacio.

Output

La salida consiste de una matriz de tamaño $n+2, m+2$. La primera fila contendrá las dimensiones de la matriz, separadas por un espacio. La siguientes líneas contienen los elementos de la matriz fila por fila todos los elementos separados por un espacio. En la columnas $m+1$ se almacena el total de la fila. En la fila $m+1$ se almacena el total de la columna.

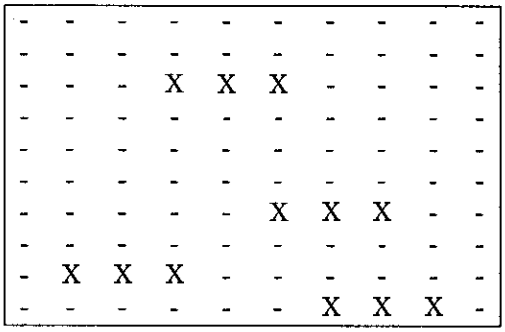
En la columnas $m+2$ se almacena el porcentaje del total de la fila con respecto al total de la suma de las columnas. La fila. En la fila $m+1$ se almacena el porcentaje con relación al total de la suma de los totales de las filas.

Mostrar los porcentajes con dos decimales y el resto de los valores sin decimales. Cada número debe imprimirse separado por un espacio. La celda $n+2, m+2$ no se imprime.

Ejemplo de entrada	**Ejemplo de salida**
1 4 4 3 8 1 8 2 6 2 4 2 3 3 3 5 5 3 4	6 6 3 8 1 8 20 0,32 2 6 2 4 14 0,23 2 3 3 3 11 0,18 5 5 3 4 17 0,27 12 22 9 19 62 1.00 0,19 0,35 0,15 0,31 1

6. Guerra de Barcos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Las guerras de barcos se juegan en una cuadrícula de papel donde cada competidor tiene que dibujar los barcos que tiene. La entrada es una cuadrícula de nxm donde en cada línea contiene $-$ y X, representando el agua y parte de un barco respectivamente. La figura siguiente representa una matriz de $10x10$ con agua y barcos. Tres caracteres X juntos representan un barco.



Los ataques se realizan con las coordenadas x, y que deben estar dentro del área descrita. Un barco se destruye cuando dos disparos han dado en el blanco. Los barcos son hileras de tres coordenadas y siempre están separados por un espacio de agua cuando están en la misma fila.

Se tiene las coordenadas de todos los ataques. Lo que se espera que indique es cuántos barcos quedan sin destruir.

Input

La primera línea de la entrada contiene la dimensión del área de juego n, m con $5 \leq m, n \leq 100$ Las siguientes m líneas contienen las filas que describen el área de juego. Cada línea solo tiene $-$ y X.

Luego viene una línea con un solo número k que indica la cantidad de ataques que ha tenido. En las líneas siguientes están las coordenadas de los ataques i, j separadas por un espacio. La coordenada superior de más a la izquierda es la $(0, 0)$, todas las coordenadas de ataques también comienzan en $(0, 0)$.

Output

La salida consiste de un solo número que indica cuantos barcos NO han sido destruídos. Un barco no es destruido si quedan dos X seguidas. Si un impacto cae en medio de un barco ya no quedaran dos X, queda destruido. Si en cambio cae un impacto a un extremo todavía tenemos dos X seguidas por lo que no se considera destruido.

Ejemplo de entrada	Ejemplo de salida
10 10 ---------- ---XXX---- ---------- ---------- ---------- -----XXX-- ---------- -XXX------ -----XXX-- ---------- 9 2 3 4 4 1 4 2 2 3 3 5 5 9 6 7 3 7 2	2

7. Matrices de Suma Máxima
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Supongamos que tenemos la matriz cuadrada definida como:

2	3	-9	6
3	4	4	-5
5	5	6	3
-1	-1	-1	10

Eliminando la primera fila y la primera columna obtenemos la matriz

4	4	-5
5	6	3
-1	-1	10

Repitiendo el proceso obtenemos

6	3
-1	10

Este proceso termina cuando obtenemos una matriz de $2x2$. Definimos la suma de la matriz como la suma de todos sus elementos.

Input

La entrada consiste de varios casos de prueba, la primera línea contiene el número de casos de prueba $n < 1000$. Para cada caso de prueba la primera línea contiene la dimensión de la matriz ($m < 100$), seguidos de m^2 elementos.

Output

Su programa debe calcular cual de estas sub matrices tiene la suma máxima. La suma máxima debe imprimirse en una línea por cada caso de prueba.

Ejemplo de entrada	**Ejemplo de salida**
3 4 -2 3 -9 6 3 -4 4 -6 5 5 6 3 -1 -1 -1 10 2 1 4 5 6 3 -1 -1 -1 -1 2 2 -1 2 2	21 16 8

8. Matrices Simétricas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Se define una matriz cuadrada como simétrica si luego de transponer las filas y columnas obtenemos la misma matriz, por ejemplo la matriz, siguiente es simétrica:

2	3
3	4

Input

La entrada consiste de varios casos de prueba, la primera línea contiene el número de casos de prueba $n < 1000$. Para cada caso de prueba la primera línea contiene la dimensión de la matriz $(2 \leq m \leq 100)$, seguidos de m^2 elementos.

Output

Su programa debe determinar, para cada caso de prueba si la matriz es simétrica, mostrando la frase Simétrica o No simétrica según sea el caso.

Ejemplo de entrada	**Ejemplo de salida**
3 4 -2 3 -9 6 3 -4 4 -6 -9 4 6 3 6 -6 3 6 2 1 4 5 6 3 1 2 3 1 1 4 3 4 1	Simetrica No simetrica No simetrica

9. Matrices de Primos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Se desea contar cuántos números comienzan y terminan con un dígito determinado. Los números primos que están en el rango $2 \leq x \leq 100000$.

Para este fin se elaborará una matriz donde las filas representan el número con el que comienza y las columnas, el número con el que termina. El contenido de la matriz representa la cantidad de primos que comienzan con un número y termina con otro.

Por ejemplo si consideramos los números del 2 al 20 los números primos son: $2, 3, 5, 7, 11, 13, 17, 19$, acomodando éstos en una matriz, obtenemos:

```
0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 1 0 1
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Donde el primo 2 comienza y termina con 2 y ha contado uno en la fila y columna $(2, 2)$. El número 13 suma 1 a la posición $(1, 3)$. Así sucesivamente.

En esta matriz se contará la cantidad de primos que comienzan (filas) y terminan(columnas) con un determinado dígito.

Input

No Hay datos de entrada

Output

La salida del programa es una matriz de $10x10$, con los datos descritos en el enunciado con la cuenta de los números primos hasta $100,000$ El ejemplo que se muestra es la matriz resultante de calcular los números hasta 10,000.

Ejemplo de entrada	**Ejemplo de salida**
	0 0 0 0 0 0 0 0 0 0 0 39 0 42 0 0 0 38 0 41 0 35 1 38 0 0 0 37 0 35 0 34 0 33 0 0 0 37 0 35 0 35 0 38 0 0 0 32 0 34 0 34 0 32 0 1 0 33 0 31 0 38 0 34 0 0 0 33 0 30 0 24 0 32 0 0 0 35 0 34 0 30 0 33 0 0 0 30 0 34 0 37 0 28 0 0 0 33 0 29

10. Batalla

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

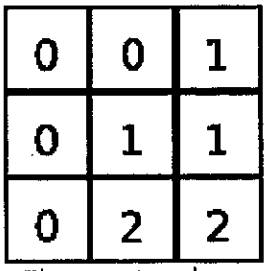
En la tierra de la ACM un gran rey mandaba y estaba muy obsesionado por el orden. El reino tenía una forma rectangular y el rey dividió el territorio en una cuadrícula rectangular pequeña que llamo *municipios*. En orden de evitar problemas después de su muerte, el rey dividió el reino entre sus hijos. Le dió a cada uno algunos *municipios* para gobernar. Sin embargo, se olvido que sus hijos desarrollaron una extraña rivalidad: El primero de sus hijos tenía aversión al segundo y no al resto. El segundo tenía aversión al tercero y no al resto, y así sucesivamente. Finalmente el último tenía resentimientos con el primero pero no con los demás hermanos.

Dada ésta extraña rivalidad entre príncipes se produjo una guerra generalizada en el reino. Cada municipio que era gobernado por un príncipe que era odiado por un hermano vecino era atacado y conquistado, (los municipios adyacentes son los que comparten una frontera vertical o horizontal).

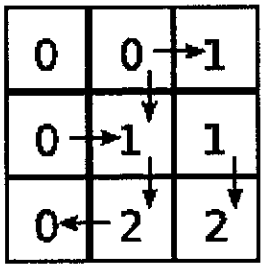
Por una regla de honor todos los ataques se realizan simultáneamente y un conjunto de ataques simultáneos se denomina *batalla*.

Usted fue contratado por un historiador para determinar cual fue la distribución final de la tierra ACM después de K batallas con las anteriores reglas

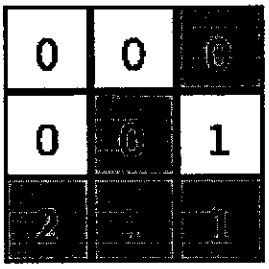
Por ejemplo (3 hermanos, llamados 0, 1 y 2):



Tierra antes de la primera Batalla



Ataques durante la primera batalla



Territorio despues de la primea batalla. Territorio ganado en gris

Input

Hay muchos casos para probar. La primera línea de cada caso de prueba contiene cuatro enteros N, R, C y K, separados por un espacio, ($2 \leq N, R, C \leq 100$, $1 \leq K \leq 100$). N es el número de hermanos, R y C son las dimensiones del reino y K es el número de batallas después de las cuales quiere conocer la distribución.

Las siguientes R líneas contienen C enteros entre $N-1$, inclusive, cada entero separado por un espacio, que representa cual hijo tiene la posesión inicial del municipio, (0 es el primer hijo, 1 es el segundo, ..., N-1 es el último).

El último caso de prueba tendrá cuatro ceros separados por un espacio.

Output

Escriba la distribución de tierra después de la $K-esima$ batalla fila por fila, en el mismo orden que se ingresó. Los enteros en las filas deben estar separados por un espacio..

Ejemplo de entrada	**Ejemplo de salida**
3 4 4 3 0 1 2 0 1 0 2 0 0 1 2 0 0 1 2 2 4 2 3 4 1 0 3 2 1 2 8 4 2 1 0 7 1 6 2 5 3 4 0 0 0 0	2 2 2 0 2 1 0 1 2 2 2 0 0 2 0 0 1 0 3 2 1 2 7 6 0 5 1 4 2 3

11. Tableros

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un patrón de tablero de ajedrez es un tablero que satisface las siguientes condiciones:

- Tiene una forma rectangular
- Contiene punto y la letra mayúscula X.
- Ninguno de los dos símbolos adyacentes, tanto horizontales y verticales son iguales
- El símbolo de la esquina inferior derecha es un punto.

Input

En la entrada existen varios casos de prueba. Cada caso de prueba consiste de dos enteros separados por un espacio. Estos estarán en el rango de 1 y 50 inclusive. El primer entero representa el número de filas del tablero a construir y el segundo el número de columnas. La entrada termina cuando no hay mas casos de prueba.

Output

La salida consiste en el tablero formado. El tablero se imprime fila por fila. Al final del tablero se imprime una fila con 10 guiones (−).

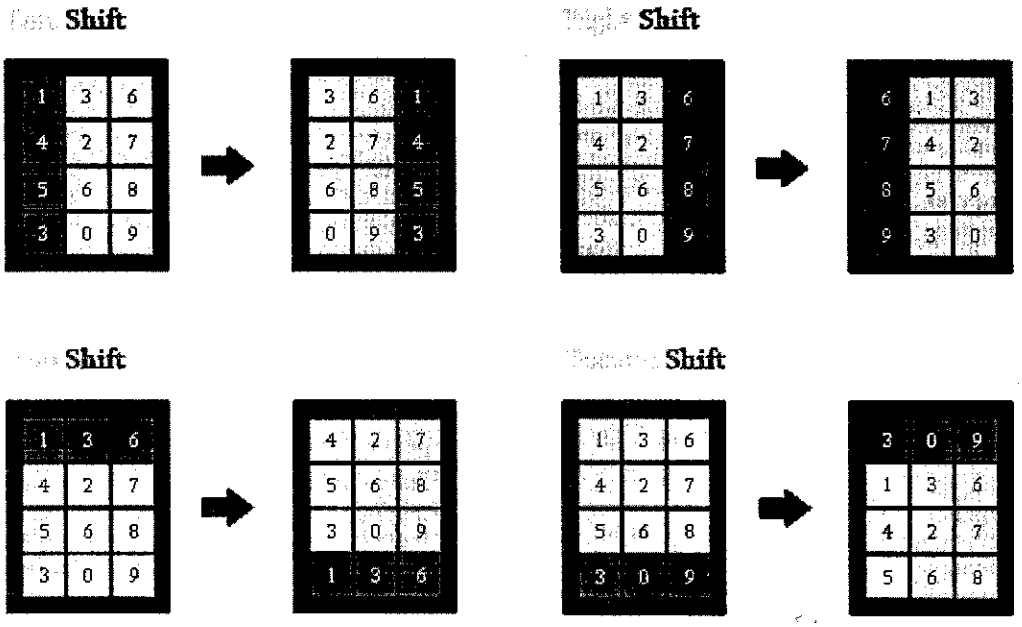
Ejemplo de entrada	**Ejemplo de salida**
8 8 1 20 1 1 2 2	.X.X.X.X X.X.X.X. .X.X.X.X X.X.X.X. .X.X.X.X X.X.X.X. .X.X.X.X X.X.X.X. ---------- X.X.X.X.X.X.X.X.X.X. ---------- . ---------- .X X. ----------

12. Desplazamientos de Filas y Columnas

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Manao tiene una matriz de NxM llena de dígitos. Las filas de la matriz están numeradas de 0 a $N-1$ comenzando con la fila superior. Las columnas están numeradas de 0 a $M-1$ comenzando en la columna de la izquierda. El quiere realizar una serie de operaciones para obtener una matriz con un valor específico en la esquina superior izquierda.

Existen cuatro tipos de desplazamientos: izquierda, derecha, arriba, abajo. Si desplazamos un lugar a la izquierda movemos todos los elementos de una columna un lugar a la izquierda y la primera columna ocupa el lugar de la última. La figura muestra los diferentes movimientos.



Si una matriz tiene una sola columna, un desplazamiento no tiene ningún efecto. Lo mismo ocurre cuando hay una sola fila.

Considere la siguiente matriz

```
1 3 6
4 2 7
5 6 8
3 0 9
```

Para llevar el 2 a la posición especificada hay que hacer un desplazamiento hacia arriba, y otro a la izquierda. La respuesta sería 2 desplazamientos.

Input

La entrada consiste en varios de datos de prueba. La primera línea de un caso de prueba tiene las dimensiones de la matriz ($1 \leq N, M \leq 50$). Luego vienen N filas con M dígitos. Después en una fila está el valor que queremos colocar en la posición superior izquierda.

La entrada termina cuando no hay más datos.

Output

La salida es un número indicando el menor número de desplazamientos a realizar para colocar este valor en la posición superior izquierda.

Ejemplo de entrada	Ejemplo de salida
4 3 1 3 6 4 2 7 5 6 8 3 0 9 2 3 4 0 0 0 0 0 0 0 0 0 0 9 9 9 1 10 0 1 2 3 4 5 6 7 8 9 7 2 3 5 5 5 5 5 5 1	2 2 3 -1

13. Tablero de Ajedrez
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un tablero de ajedrez en general tiene una dimensión de $8x8$, se caracteriza de la siguiente manera las filas están marcadas por números, del 1 al 8, y las columnas están marcadas por las letras, 'a'- 'h'. Una celda está descrita por su marca de la columna y luego su marca de fila, como "e4". Mientras si se trabaja en un programa de ajedrez, es necesario convertir esta descripción en su equivalente denominada número interior de la celda. Internamente las celdas están numeradas fila por fila del 1 a 64 en su programa, es decir, la celda "a1"tiene el número 1, la celda "b1"tiene el número 2, la celda "c1"tiene el número 3, así hasta, la celda "h8"tiene el número 64. Dada una celda de ajedrez, que describe bien la marca de celda o el número interno de celda, cambiar la notación (es decir, si te dan la marca usted necesita para devolver el número interno de celda, y viceversa).

Input

La entrada de datos contendrá ya sea marca de celda o número interno de celda. Si la entrada contiene marca de celda entonces contendrá exactamente dos caracteres la primera ('a'- 'h') y la segunda ('1'- '8'), y si la entrada es número interno de celda entonces el valor estará entre 1 y 64, la entrada termina cuando no haya más casos de prueba.

Output

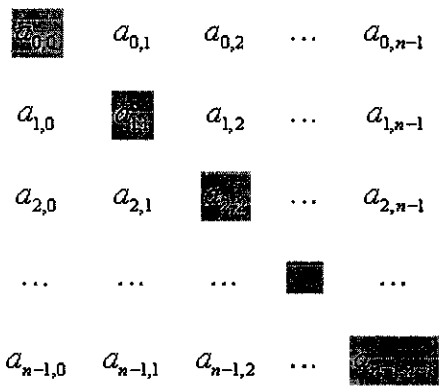
Para cada caso de entrada mostrar la equivalencia en la otra notación es decir si la entrada es número de celda entonces mostrar su equivalente número interno de celda, y viceversa.

Ejemplo de entrada	**Ejemplo de salida**
1	a1
2	b1
26	b4
c1	3
e4	29
h8	64

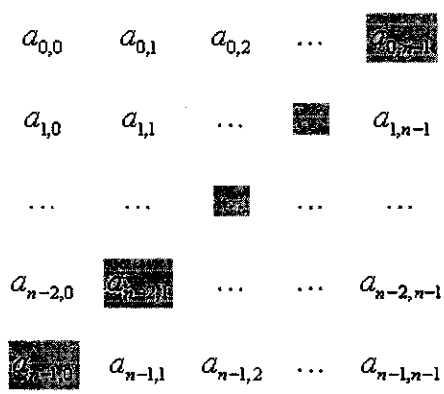
14. Desproporción Diagonal
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Se le ha encargado a usted calcular la desproporción diagonal de una matriz cuadrada. La desproporción diagonal de una matriz cuadrada es la suma de los elementos de su diagonal principal menos la suma de los elementos de la diagonal secundaria o colateral.

La figura muestra la diagonal principal.



La figura muestra la diagonal secundaria.



Input

La entrada de datos consiste de varios casos de prueba. El primer número tiene el número de casos de prueba.

Cada caso de prueba comienza con un número entero que tiene el número de filas de la matriz. Seguidamente vienen N líneas, con la misma cantidad de caracteres, que contienen caracteres entre 0 y 9.

Output

La salida es una línea por caso de prueba que tiene diferencia de las dos diagonales.

Ejemplo de entrada	Ejemplo de salida
4 3 190 828 373 4 9000 0120 0000 9000 1 6 10 7748297018 8395414567 7006199788 5446757413 2972498628 0508396790 9986085827 2386063041 5687189519 7729785238	1 -1 0 -24

Capítulo 8

Métodos, funciones, procedimientos y clases

8.1. Definición

Volvamos a los conceptos presentados el capítulo 1 cuando se construyó el primer programa.

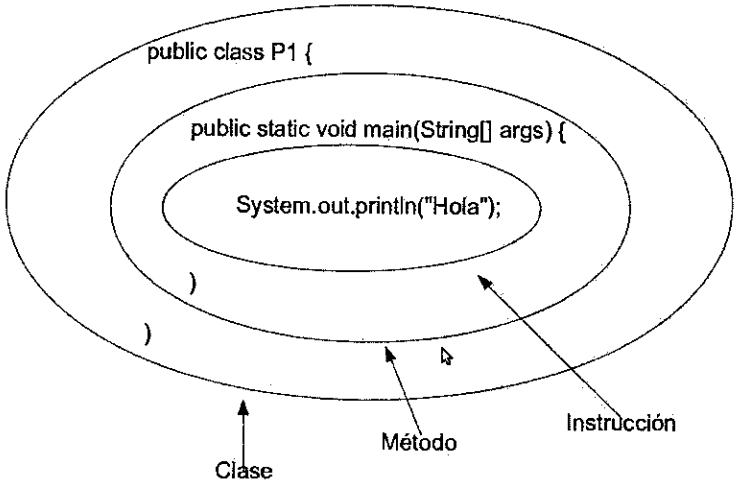


Figura 8.1: Estructura de un programa Java

8.2. Herramientas de desarrollo

En la imagen 8.1. vemos que se ha definido un método en el interior de una clase.

En java una clase es un archivo que tiene un nombre y la instrucción *public class Nombre* donde el nombre del archivo debe igualar al nombre de la clase. Por ejemplo:

```
public class P1 {
     instrucciones de la clase
     }
```

Se pueden definir dos tipos de clases: La que puede ejecutarse en forma independiente y la que requiere que otra clase la invoque. Para que una clase pueda ejecutarse se requiere que tenga la instrucción:

```
public static void main(String[] args) {
       instrucciones
       }
```

Estas instrucciones definen un método denominado *main*. El nombre *main* indica que este método se ejecutará cuando se inicie el programa. Entre paréntesis están los argumentos que puede aceptar. En este caso un vector de tipo cadena que se llama *args*. Estos parámetros son los que se pueden incluir en la línea de comando al ejecutar el programa. Por ejemplo:

```
c:> java P1 los parámetros deseados
```

En este caso recibirá un vector de tipo cadena con tres valores *args[0]=los, args[1]=parámetros, args[2]=deseados.*

- Denominamos *funciones* a los métodos que devuelven un resultado o valor.
- Denominamos *procedimientos* a los métodos que devuelven un valor

El método *main* es un procedimiento porque no devuelve un valor. En java solo nos referimos como métodos.

¿Donde se pueden escribir los métodos? Es posible codificar métodos en el programa principal, es decir, el que incluye el método *main*, o en un archivo separado. Inicialmente veamos como se trabaja en programa principal.

```
public class Hola {

    //antes de un mÃ©todo
    public static void main(String[] args) {
        System.out.print(''hola");
    }
    // despues de un metodo
}
```

En el ejemplo vemos que se escribe antes o después de un método.

8.2.1. Sintaxis para escribir funciones

Para escribir una función se sigue la siguiente sintaxis:

```
tipoAdevolver NombreDeLaFunción (tipo NombeDelParametro, ....){
    intrucciones
    return valor
  }
```

Por ejemplo se deseamos realizar una función que nos devuelva *true* cuando un número es par y *false* cuando es impar, primero le damos un nombre adecuado, por ejemplo *esPar*. Luego decidimos que parámetros va a recibir. En nuestro caso decidamos que pasaremos un número entero. El código será simplemente verificar el último bit si es cero devolvemos true en otros casos false. Con estos datos el programa queda como:

```
boolean esPar(int n){
   if ((n&1)==0)
       return true;
   else
       return false;
}
```

Para utilizar la función es suficiente con insertar el nombre definido como si fuera una instrucción parte del lenguaje. Para ejemplificar este método construyamos un programa para sumar todos los números pares entre entre uno y quince.

```
public class F1 {

    static boolean esPar(int n){
        if ((n&1)==0)
            return true;
        else
            return false;
    }
    public static void main(String[] args) {
        int suma=0;
        for (int i=0;i<16;i++)
            if(esPar(i))
            suma+=i;
        System.out.println(suma);
    }
}
```

Es necesario aclarar que el método *main* es de tipo *static*, por esto es necesario incluir la palabra *static* antes de la definición de la función *esPar*. Esto significa que la dirección de memoria donde se cargará para su ejecución es fija. No cambia de una ejecución a otra.

8.2.2. Sintaxis para escribir procedimientos

Para escribir procedimientos seguimos los mismos pasos que para escribir funciones. Lo que no se hace es incluir un valor a retornar por los que no tiene un tipo.

```
void NombreDelProceimiento (tipo NombeDelOarametro, ....){
intrucciones
}
```

8.3. Variables locales y globales

Las variables pueden ser **globales** o **locales**. Las variables globales son visibles en todo el programa, las variables locales solo en el método que se definió. Veamos el siguiente programa que contiene una función para sumar los elementos de un vector:

```
public class F2 {
    static int Sumar(int [] v){
        int suma=0;
        for (int i=0;i<v.length;i++)
            suma+=v[i];
        return suma;
    }
    public static void main(String[] args) {
        int suma=0;
        int [] v = {1,2,3,4,5,6};
        suma=Sumar(v);
        System.out.println(suma);
    }
}
```

Se definieron dos variables con el nombre *suma* una en el método y otra en el programa principal. Cada una se considera diferente y local, por que su visibilidad esta dada solo en el método que se definió.

Para ejemplificar las variables globales definiremos una variable fuera de los métodos, esto hasta que su visibilidad sea en todos. El programa siguiente muestra como hacer esto. Otra vez tuvimos que agregar la palabra *static*, porque el programa principal es de este tipo.

```
public class F3 {

    static int suma=0;
    static void Sumar(int [] v){
        for (int i=0;i<v.length;i++)
            suma+=v[i];
    }
    public static void main(String[] args) {
        int [] v = {1,2,3,4,5,6};
        Sumar(v);
        System.out.println(suma);
    }
}
```

El problema que tienen las variables globales se puede resumir en la falta de control. Cualquier método puede modificar su valor, y esto puede llevar a errores difíciles de detectar.

8.4. Definición de métodos en archivos separados

Para definir métodos en archivos separados del programa principal, primero se crea una clase de tipo *public* y en el interior de la clase colocamos los métodos. Para ejemplificar esto hagamos que el método para sumar los elementos de un vector sea una clase separada. El programa queda como sigue:

```
public class Suma {
     int Sumar(int [] v){
        int suma=0;
        for (int i=0;i<v.length;i++)
            suma+=v[i];
        return suma;
    }
}
```

Para utilizar este programa debemos crear una instancia del mismo. En este instancia serán accesibles los métodos definidos. El código que debemos escribir es el siguiente:

```
public class P2 {
    public static void main(String[] args) {
        Suma a1=new Suma();
        int [] v = {1,2,3,4,5,6};
        int suma = a1.Sumar(v);
        System.out.println(suma);
    }
}
```

En este código no fue necesario especificar *static*. Esto debido a que la dirección donde se encuentran las instrucciones ejecutables se definió con la instrucción: *Suma a1=new Suma()*. Ahora se tiene la instancia *a1* los métodos son accesibles por su nombre. En el ejemplo para acceder al método que suma los elementos del vector escribimos *a1.Sumar(v)*, como el resultado es un entero que lo asignamos a una variable para luego mostrar en pantalla.

Para asignar valores iniciales cuando se define una clase, se debe crear un **constructor**. Un constructor se define como un método que tiene el mismo nombre que el nombre de la clase.

Cambiemos el programa para hallar la suma de elementos de un vector. Creamos un constructor donde se pasa el vector. Luego en los métodos ya no le pasamos el vector porque ya lo definimos con anterioridad. El programa queda como sigue:

```
public class vect {
    int [] v;
    public vect(int[] v) {
        this.v = v;
    }

    int Sumar(){
        int suma=0;
        for (int i=0;i<v.length;i++)
            suma+=v[i];
        return suma;
```

```
    }
}
```

Es necesario explicar la palabra *this*. En la definición del constructor una alternativa es:

```
public vect(int[] v2) {
v = v2;
}
```

En el ejemplo pusimos el mismo nombre a la variable del parámetro $v2$ que a la variable de la clase v para diferenciar a cual variable nos referimos se utiliza *this*, que indica que nos referimos a la variable definida en la clase y no a la de los parámetros.

Es posible definir múltiples constructores con el mismo nombre, siempre y cuando los parámetros con los que se invoque el método sean diferentes. Esto se llama sobrecarga de métodos. ¿Cuál constructor se utilizará? El que iguale en nombre y parámetros en la llamada y definición del mismo.

8.5. Ejemplos de aplicación

Un mecanismo para contar los números primos *Contando primos* es el algoritmo denominado criba de Eratóstenes. Este algoritmos indica que una criba de números primos se construye como sigue:

1. Se marcan los múltiplos de 2.
2. Luego los de 3, 5, 7, etc. sucesivamente.
3. Una vez completado el marcado los no marcados son los números primos.

Este algoritmo esta descrito en el capítulo de números primos con mayor detalle. Queremos contar todos los números primos entre a y b. EL programa inicial puede ser el siguiente:

```
public class P1 {

    public static void main(String[] args) {
        int n = 10000000;
        int[] criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
             // si no se marco es numero primo
        if (criba[i] == 0) {
             //marcar los multiplos de i
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }

        int contar = 0, a=0, b=1000;
        for (int i = a; i <= b; i++) {
            if (criba[i]==0)
                contar++;
        }
        System.out.println(contar);
```

```
    }
}
```

Los objetivos que queremos alcanzar son los siguientes:

1. Escribir una clase que pueda utilizarse en múltiples programas.
2. Construir métodos para contar los primos,
3. Construir métodos para construir la criba
4. Especificar el tamaño de la criba.

Estos objetivos los iremos desarrollando paso a paso. Primero creamos una clase que la denominaremos *Criba*. Llevamos todo el código a esta clase.

```
public class P2 {
    public P2() {
        int n = 10000000;
        int[] criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }

        int contar = 0, a=0, b=1000;
        for (int i = a; i <= b; i++) {
            if (criba[i]==0)
                contar++;
        }
        System.out.println(contar);
    }
}
```

Para invocar al mismo el programa principal usamos la instrucción *Criba a=new Criba()*

Este programa tiene el problema que solo sirve para un caso particular. Dividiremos el mismo en constructor y un método para contar que devuelva el número de primos. Para hacer esto el vector *Criba* cambiamos para que sea una variable global, para que este disponible en los diferentes métodos.

```
public class Criba {

    int[] criba;
    public Criba() {
        int n = 10000000;
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
```

```
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }
    }
    public int Contar(int a, int b) {
        int contar = 0;
        for (int i = a; i <= b; i++) {
            if (criba[i]==0)
                contar++;
        }
        return contar;
    }
}
```

Adicionalmente se han agregado dos parámetros para que la cuenta sea en el rango solicitado.

Para hacer más entendible el programa, crearemos un método *esPrimo* que devuelva verdadero o falso.

```
public class Criba {

    int[] criba;
    public Criba() {
        int n = 10000000;
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }
    }
    public boolean esPrimo(int i) {
        if (criba[i]==0)
            return true;
        else
            return false;
    }
    public int Contar(int a, int b) {
        int contar = 0;
        for (int i = a; i <= b; i++) {
            if (esPrimo(i))
                contar++;
        }
        return contar;
    }
}
```

Es más fácil de entender porque no es necesario conocer la implementación $criba[i] == 0$ para probar la primalidad.

El programa desarrollado crea una criba de tamaño fijo de $n = 10000000$ y es deseable crear una criba variable para reducir el tamaño de memoria utilizado y el tiempo de crear la criba. Para esto creamos un constructor que reciba el parámetro n. Con esto tendremos dos constructores, uno cuando se especifica el tamaño y otro para el valor máximo cuando no se especifica el tamaño:

```
public class Criba {

    int[] criba;
    public Criba() {
        int n = 10000000;
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }
    }

    public Criba(int n) {
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }
    }
    public boolean esPrimo(int i) {
        if (criba[i]==0)
            return true;
        else
            return false;
    }
    public int Contar(int a, int b) {
        int contar = 0;
        for (int i = a; i <= b; i++) {
            if (esPrimo(i))
                contar++;
        }
        return contar;
    }
}
```

Hemos creado un nuevo problema, código repetido. Ahora crearemos un método *crearCriba* donde se hallarán los valores y eliminará el código repetido.

```
public class Criba {

    int[] criba;
    public Criba() {
        crearCriba(10000000);
    }
```

```
    public Criba(int n) {
        crearCriba(n);
    }
    public void crearCriba(int n) {
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (criba[i] == 0) {
            for (int j = i + i; j <= n; j = j + i) {
                criba[j] = 1;
            }
        }
    }
    public boolean esPrimo(int i) {
        if (criba[i]==0)
            return true;
        else
            return false;
    }
    public int Contar(int a, int b) {
        int contar = 0;
        for (int i = a; i <= b; i++) {
            if (esPrimo(i))
                contar++;
        }
        return contar;
    }
}
```

Ahora trabajemos en el código del método *crearCriba*. El código $criba[i] == 0$ se cambia por $esPrimo(i)$. Luego creamos un método *marcarMultiplos* que marque los múltiplos de i. Con esto llegamos a la versión final:

```
public class Criba {

    int[] criba;
    public Criba() {
        crearCriba(10000000);
    }

    public Criba(int n) {
        crearCriba(n);
    }
    public void crearCriba(int n) {
        criba = new int[n + 1];
        // construccion de la criba
        for (int i = 2; i <= n; i++)
            if (esPrimo(i))
            marcarMultiplos(i);
    }
    public void marcarMultiplos(int i) {
        for (int j = i + i; j <= criba.length; j = j + i) {
            criba[j] = 1;
```

```
        }
    }
    public boolean esPrimo(int i) {
        if (criba[i]==0)
            return true;
        else
            return false;
    }
    public int Contar(int a, int b) {
        int contar = 0;
        for (int i = a; i <= b; i++) {
            if (esPrimo(i))
                contar++;
        }
        return contar;
    }
}
```

8.6. Cuando hay que usar métodos y clases

Es de interés dividir el programa en métodos para:

1. Hacerlo más fácil de modificar. Cada vez que se realizan cambios en los programas se hace más difíciles de entender.
2. Hacerlo más fácil de entender. Es más fácil de entender un parte de un programa que todo el conjunto, esto facilita el mantenimiento.
3. Mejorar su diseño.
4. Eliminar el código duplicado. Eliminar la redundancia, las correcciones se hacen en un solo lugar.
5. Encontrar errores. Al reorganizar un programa se pueden apreciar tódas las suposiciones que se hicieron.
6. Para programar más rápido. Utilizando métodos ya desarrollados, se desarrollará más rápidamente un nuevo programa.
7. Distribuir un programa. Cuando distribuimos un programa para que pueda ser incluido en otras aplicaciones, se distribuyen las clases con las instrucciones de uso.

Estos cambios se hacen creando métodos, funciones, procedimientos y clases.

8.7. Ejercicios

Codificar los siguientes ejercicios tomando en cuenta todos los aspectos mencionados en el capítulo. Eliminar todo el código repetido. Crear una clase matriz que implemente los siguientes métodos:

1. Crear una clase matriz que implemente los siguientes métodos:

   a) Definir una matriz bidimensional
   b) Transponer la matriz
   c) Sumar con otra matriz pasada como parámetro.
   d) Multiplicar con otra matriz pasada como parámetro.
   e) Imprimir la matriz resultante por filas
   f) Copiar la matriz a otra pasada como parámetro.

2. Crear una clase que permita ordenar una matriz bidimensional. Se deben implementar dos algoritmos de ordenación, los métodos a implementar son:

   a) Definir una matriz bidimensional
   b) Ordenar por la primera columna.
   c) Ordenar por una columna especificada.

Capítulo 9

Números Primos

9.1. Introducción

Se hace muy importante la teoría de números en el aprendizaje de la programación. Este conocimiento permite comprender los aspectos que hacen al proceso computacional, tales como las capacidades de la máquina, el tamaño máximo de números que se pueden tratar y como trabajar con números más grandes.

En el tratamiento de este capítulo se toman en cuenta los aspectos relativos al tiempo de proceso para hacer eficiente los algoritmos expuestos.

9.2. Variables del lenguaje Java

El lenguaje de programación Java tiene varios tipos de variables enteras que se muestran en el cuadro 9.1.

tipo	Nro. Bytes	Nro. Bits	Rango de números permitido
byte	1 bytes	8 bits	*desde* $-(2^7+1)$ *hasta* $+2^7$
short	2 bytes	16 bits	*desde* $-(2^{15}+1)$ *hasta* $+2^{15}$
int	4 bytes	32 bits	*desde* $-(2^{31}+1)$ *hasta* $+2^{31}$
long	8 bytes	64 bits	*desde* $-(2^{63}+1)$ *hasta* $+2^{63}$

Cuadro 9.1: Tipos de variables enteras en Java

Para evaluar el tiempo de proceso de las operaciones de suma, multiplicación y división construimos un programa que repita muchas veces una operación, midiendo el tiempo de ejecución. Esto permite determinar cuanto tiempo consumen los diferentes tipos de variables y escoger la más adecuada para mejorar el tiempo de proceso. El tiempo que tomó la operación de multiplicar realizando 10.000.000 de operaciones de multiplicación para una variable entera.

Una vez procesado este programa construimos el cuadro 9.2 para comparar el tiempo de proceso para cada tipo de variable. Se puede observar que una operación de división demora mucho más que una operación de multiplicación.

En el sistema operativo Linux se utiliza la instrucción *time* para medir el tiempo de proceso. Esta instrucción de la línea de comandos nos da el tiempo de ejecución total del programa.

Operacion	tipo	Tiempo en mseg para 10.000.000 repeticiones
suma	int	110
suma	long	190
multiplicación	int	140
multiplicación	long	230
división	int	771
división	long	2334

Cuadro 9.2: Comparación de los tiempos de ejecución de las operaciones básicas

9.3. Definición

La teoría de los números primos ha tenido un desarrollo muy intenso con las aplicaciones de criptografía. En esta sección le dedicamos un especial interés debido a las complejidades que lleva éste cuando se trabaja con números grandes.

Se define como primo el número entero positivo que es divisible solamente por 1 o por si mismo. Los primeros números primos son $2, 3, 5, 7, 11, 19...$ Como se ve el único número primo par es el número 2. El número 1 no es parte de los números primos, sin embargo, algunos autores amplían la definición para incluirlo.

Para muchas aplicaciones es necesario realizar un test de primalidad que significa verificar si el número es primo o no. Esto se puede realizar por divisiones sucesivas sin embargo no es un método muy adecuado cuando se trata de números grandes.

Es posible probar la primalidad de un número n en un tiempo proporcional a $O(\sqrt{n})$ con el siguiente código de divisiones sucesivas

```
j = 2;
while (j * j <= n) {
  if ((n%j)==0)
      break; //no primo
      j++;
}
```

Ahora escribamos un programa que cuente los números primos hasta 10,000,000 y midamos el tiempo de proceso.

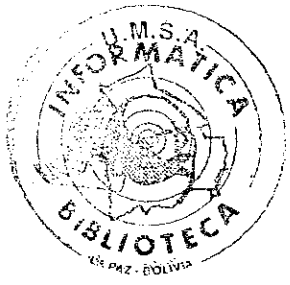
```
/**
* Solucion al problema Contando primos
* por Divisiones Sucesivas
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class DivisionesSucesivas{
    public static void main(String[] args) {

        int contar = 0;
        int n = 10000000;
        for (int i = 2; i <= n; i++) {
            if (EsPrimo(i))
                contar++;
```

```
        }
        System.out.println(contar);
    }

    static boolean EsPrimo(int numero){
        for (int j=2;j * j <= numero;j++) {
            if ((numero% j)==0)
                return false; //no primo
        }
        return true; //primo
    }
}
```



Analizando los números primos vemos que todos los números que terminan en $2, 4, 6, 8, 0$ son divisibles por 2 y por lo tanto no son primos. Los que terminan en 5 son divisibles por 5, y tampoco son primos. El programa corregido es el siguiente:

```
/**
* Solucion al problema Contando primos
* por Divisiones Sucesivas Mmejorado
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class DivisionesSucesivasMejorado{
    public static void main(String[] args) {
        int contar = 4;
        int n = 10000000;
        for (int i = 11; i <= n; i+=2) {
            if (EsPrimo(i))
                contar++;
        }
        System.out.println(contar);
    }

    static boolean EsPrimo(int numero){
        if (numero% 3==0)
            return false;
        if (numero% 5==0)
            return false;
        for (int j=3;j * j <= numero;j=j+2) {
            if ((numero% j)==0)
                return false; //no primo
        }
        return true; //primo
    }
}
```

Múltiplo de	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2			x		x		x		x		x		x	
3					x			x			x			x
5									x					x
Marcados			x		x		x	x	x		x		x	x

Cuadro 9.3: Ejemplo de una criba de Eratóstenes

9.4. Generación de primos

El problema que representa el método presentado de divisiones sucesivas es la cantidad de divisiones que hay que realizar. Como ya vimos el tiempo que toma una división es mucho mayor al de la suma. Para convertir éstas divisiones en suma de números, la forma más fácil es a través del método denominado la criba de Eratóstenes.

La criba de Eratóstenes se contruye a partir de los múltiplos de los números como sigue

1. Se marcan los múltiplos de 2.
2. Luego los de 3, 5, 7, etc. sucesivamente.
3. Una vez completado el marcado los no marcados son los números primos.

Como se observa en el cuadro 9.3 se marcan en la primera pasada los múltiplos de 2, la segunda pasada los múltiplos de 3 y así sucesivamente. Al terminar el proceso los dígitos no marcados son los números primos. Codificando este programa el tiempo de proceso se reduce al eliminar las divisiones.

```
/**
* Solucion al problema Contando primos
* utilizando una criba
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class Criba {

    public static void main(String[] args) {
        int n = 10000000;
        short[] criba = new short[n + 1];
        Criba(criba, n);
        int contar = 0;
        for (int i = 2; i <= n; i++) {
            if (criba[i]==0)
                contar++;
        }
        System.out.println(contar);
    }

    static void Criba(short[] p, int n){
        int i = 2, j = 0, a, b;
        // construccion de la criba
        for (i = 2; i <= n; i++)
            if (p[i] == 0) {
            for (j = i + i; j <= n; j = j + i) {
```

```
                p[j] = 1;
            }
        }
    }
}
```

Este código tiene la limitación de solo poder crear un vector hasta 100, 000, 000 números enteros de 16 bits. Es posible crear una criba con 100, 000, 000 utilizando una estructura de bits. El siguiente programa muestra como contar cuantos números primos hay hasta 100.000.000 .

```
import java.util.*;

/**
* Programa contar los numeros primos hasta = 100,000,000
* utilizando la Criba de Eratostenes usaando BitSet
*
* @author Jorge Teran
*/
public class CribaBitSet {
     public static void main(String[] s) {
         int n = 100000000;
         BitSet a = new BitSet(n + 1);
         int contar = 0;
         int i = 2, j = 0;
         // construccion de la criba
         for (i = 2; i * i <= n; i = i + 1) {
             if (!a.get(i)) {
                 for (j = i + i; j <= n; j = j + i) {
                     a.eet(j);
                     }
                 }
             }
         // contar los numeroe primos
         for (i = 2; i <= a; i++) {
             if (!a.get(i)) {
                 contar++;
                 // System.out.print(i+'' ");
                 }
             }
         System.out.println(contar + '' primos");
         }
}
```

Los números primos que pueden calcularse a través de este método se consideran números pequeños. El número máximo depende de la memoria de la máquina, la implementación realizada y el lenguaje de programación. Vea que si almacena solo los números impares puede almacenar el doble de números en la criba.

9.5. Criba de Atkin

Conjuntamente Arthur Oliver Lonsdale Atkin conjuntamente con Daniel J. Bernstein han desarrollado la criba de Atkin. Este trabajo se basa en la teoría: *cribas utilizando formas cuadráticas binarias*, que publicaron el año 2003. Este algoritmo es más complejo que la criba de Eratóstenes, que desarrollo su algoritmos el siglo III antes de Cristo.

El algoritmo no se explicara en éste texto pero se basa en las formas cuadráticas $4x^2+y^2$ y $3x^2+y^2$. Se seja para el lector el análisis del mismo. El programa java que usa el la criba de Atkin es:

```
import java.util.ArrayList;
public class Atkin {
    /**
     * Solucion al problema Contando primos
```

```
     * utilizando una criba de Atkin
     *
     * @author Jorge Teran
     **/
    public static void main(String[] args) {

        int limite=100000000;

        boolean[] esPrimo = new boolean[limite + 1];
        int sqrt = (int)Math.sqrt(limite);

        for (int x = 1; x <= sqrt; x++)
            for (int y = 1; y <= sqrt; y++)
            {
            int n = 4 * x * x + y * y;
            if (n <= limite && (n% 12 == 1 || n% 12 == 5))
                esPrimo[(int)n] ^= true;

            n = 3 * x * x + y * y;
            if (n <= limite && n% 12 == 7)
                esPrimo[n] ^= true;

            n = 3 * x * x - y * y;
            if (x >y && n <= limite && n% 12 == 11)
                esPrimo[n] ^= true;
        }

        for (int n = 5; n <= sqrt; n++)
            if (esPrimo[n])
            {
            int s = n * n;
            for (int k = s; k <= limite; k += s)
                esPrimo[k] = false;
        }



        int contar=2;
        for (int n = 5; n <= limite; n+=2)
            if (esPrimo[n])
            contar++;
        System.out.println(contar);

    }
}
```

La página http://primes.utm.edu/ proporciona un acceso importante a la teoría de números primos y publica la cantidad de primos por rango conocidos a la fecha, están detallados en el cuadro 9.4.

Hasta	Cantidad de Primos
10	4
100	25
1.000	168
10.000	1.229
100.000	9.592
1.000.000	78.498
10.000.000	664.579
100.000.000	5.761.455
1.000.000.000	50.847.534
10.000.000.000	455.052.511
100.000.000.000	4.118.054.813
1.000.000.000.000	37.607.912.018
10.000.000.000.000	346.065.536.839
100.000.000.000.000	3.204.941.750.802
1.000.000.000.000.000	29.844.570.422.669
10.000.000.000.000.000	279.238.341.033.925
100.000.000.000.000.000	2.623.557.157.654.233
1.000.000.000.000.000.000	24.739.954.287.740.860
10.000.000.000.000.000.000	234.057.667.276.344.607
100.000.000.000.000.000.000	2.220.819.602.560.918.840
1.000.000.000.000.000.000.000	21.127.269.486.018.731.928
10.000.000.000.000.000.000.000	201.467.286.689.315.906.290
100,000,000,000,000,000,000,000	1,925,320,391,606,803,968,923
1,000,000,000,000,000,000,000,000	18,435,599,767,349,200,867,866

Cuadro 9.4: Cantidad de primos por rango conocidos a la fecha

9.6. Factorización

La factorización de un número consiste en descomponer el mismo en sus divisores. Se demuestra que todos los factores son números primos. Analice que si uno de sus factores es un número compuesto también podría descomponerse en factores. Para una demostración más formal vean [THCR90].

La forma de expresar éstos factores es como sigue: $a_1^{n_1} a_2^{n_2} a_3^{n_3} \ldots$. Los factores primos de 168 son $2, 2, 2, 3, 7$ que se expresa como $2^3 \cdot 3 \cdot 7 = 168$. El siguiente código muestra el proceso de factorización de números. Hay que indicar que este proceso es aplicable a números pequeños.

```
/**
* Calcula los factores primos de N
* Ejemplo java Factors 81
*
* @author Jorge Teran
**/

public class Factores {

     public static void main(String[] args) {
         long N = Long.parseLong(args[0]);
         //long N = 168;
         System.out
         .print(``Los factores primos de: "
         + N + `` son: ");

         for (long i = 2; i <= N / i; i++) {

             while (N% i == 0) {
                 System.out.print(i + `` ");
                 N = N / i;
                 }
             }

         // Si el primer factor ocurre uña sola vez
         if (N >1)
             System.out.println(N);
         else
             System.out.println();
         }
}
```

9.7. Prueba de la primalidad por división entre primos

Como analizamos en la sección anterior es suficiente con dividir los números solo por números primos para probar su primalidad. Para esto crearemos un vector de primos e iremos dividiendo entre los primos generados que se almacenaran en un vector.

Iniciamos un vector con el primer número primo que es el 2 luego en un ciclo probamos el número 3, dividiendo entre los números primos generados. Como resulta ser primo lo agregamos al vector y continuamos con un siguiente número. Este Proceso también reduce la cantidad de divisiones, dado que solo dividimos por números primos menores la raíz cuadrada del número que queremos probar.

El siguiente programa muestra esta implementación.

```
import java.util.ArrayList;


public class DivideSoloPrimos {
    static int max=10000000;
    static int [] p = new int[max];

    public static void main(String[] args) {
        p[0]=2;p[1]=3;p[2]=5;p[3]=7;p[4]=11;
        int i=0, j=0, l=4;
        for (i = 11; i <= max; i+=2)
            if (EsPrimo(i)){
            p[++l]=i;
        }


        System.out.println(''Ultimo numero "+p[l]+'' total de numeros "+(l));
    }
    static boolean EsPrimo(int numero){
        if ((numero & 1)==0)
            return false;
        if ((numero%5 ==0))
            return false;
        for (int j=0;(p[j] * p[j]) <= numero;j++) {
            if ((numero                return false; //no primo
        }
        return true; //primo
    }
}
```

Cuántos números primos podemos calcular con estos métodos?. Las limitaciones están en la cantidad de memoria que se utiliza y la implementación. Esto con variable de memoria de 64 bits (long) el número grande que se pueda hallar estará dado por éste valor.

9.8. Prueba de la primalidad

La prueba de primalidad es simple para números primos pequeños. Se puede hacer por divisiones sucecivas, aún cuando, es mejor utilizar una criba con números precalculados. Las dificultades en el cálculo de números primos radica cuando los números a tratar son grandes. Esto hace necesario buscar otros métodos para determinar la primalidad de los mismos. Analizamos algunos métodos sin ahondar en las demostraciones que están ampliamente desarrolladas en la teoría de números y el área de criptografía.

9.9. Teorema de Fermat

Este teorema indica que todo número primo cumple la relación:

$$a^{n-1} \equiv 1 \mod n \, \forall a \leq n \tag{9.9.1}$$

por lo que parece suficiente realizar este cálculo para determinar la primalidad.

Desafortunadamente solo sirve para conocer si un número es compuesto dado que para algunos números primos no se cumple. Estos números se denominan números de Carmichael . Veamos por ejemplo el número compuesto 561:

$$2^{561-1} \mod 561 = 1$$

Algunos números que no cumplen este teorema son el 561, 1105, 1729. Estos números se denominan pseudo primos.

Para determinar si un número es compuesto podemos comenzar un algoritmo con $a = 2$ y el momento que no se cumpla el teorema podemos decir que el número es compuesto. En otros casos se indica que el número es probablemente primo.

La utilidad de este método se ve en el tratamiento de números grandes donde la facilidad de la exponenciación módulo es mucho más eficiente que la división.

9.10. Prueba de Miller - Rabin

Este algortimo tiene su nombre por los autores del mismo. Esta prueba provee un algoritmo probabílistico eficiente aprovechando algunas características de las congruencias.

Dado un entero n impar, hagamos $n = 2^r s + 1$ con s impar. Escogemos un número entero aleatoriamente y sea éste $1 \leq a \leq n$. Si $a^s \equiv 1 \mod (n)$ o $a^{2^j s} \equiv -1 \mod (n)$ para algún j, que esté en el rango $0 \leq j \leq r - 1$, se dice que n pasa la prueba. Un número primo pasa la prueba para todo a.

Para utilizar este concepto se escoge un número aleatorio a. Si pasa la prueba, probamos con otro número aleatorio. Si no pasa la prueba decimos que el número es compuesto. Se ha probado que la probabilidad de que, un número a pase la prueba, es de $\frac{1}{4}$ por lo que hay que realizar varias pruebas para tener más certeza.

El propósito de comentar este algoritmo es el hecho que la implementación de Java ya lo incluye en sus métodos para trabajar con números grandes por lo que, no desarrollaremos el algoritmo en extenso.

Con la finalidad de comparar con los otros algoritmos se incluye el programa.

```
/**
* Emeplo de una implementacion de algoritmo
* de Miller - Rabin
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class MillerRabin {

    public static int n = 10000000;

    public static void main(String[] args) {
        int contar = 0;
        for (int i = 2; i <= n; i++)
            if (Miller_Rabin(i) == true)
            contar++;
        System.out.println(contar);
    }

    public static boolean Miller_Rabin(int n) {
```

```
        if (n <= 1)
            return false;
        else if (n == 2)
            return true;
        else if (iteracion_Miller_Rabin(2, n)
             && (n <= 7 || iteracion_Miller_Rabin(7, n))
         && (n <= 61 || iteracion_Miller_Rabin(61, n)))
        return true;
        else
            return false;
    }

    private static boolean iteracion_Miller_Rabin(int a, int n) {
        int d = n - 1;
        int s = Integer.numberOfTrailingZeros(d);
        d >>= s;
        int a_elevado_a = exponente_modulo(a, d, n);
        if (a_elevado_a == 1)
            return true;
        for (int i = 0; i <s - 1; i++) {
            if (a_elevado_a == n - 1)
                return true;
            a_elevado_a = exponente_modulo(a_elevado_a, 2, n);
        }
        if (a_elevado_a == n - 1)
            return true;
        return false;
    }

    private static int exponente_modulo(int base, int potencia, int modulo) {
        long resultado = 1;
        for (int i = 31; i >= 0; i--) {
            resultado = (resultado * resultado) % modulo;
            if ((potencia & (1 << i)) != 0) {
                resultado = (resultado * base) % modulo;
            }
        }
        return (int) resultado;
    }

}
```

9.11. Tiempo de proceso de los algoritmos presentados

Para verificar experimentalmente la eficiencia de los algoritmos presentados realizamos la ejecución de los mismos, para 100,000,000 de números. El resultado se muestra en el cuadro 9.5.

Podemos ver en el cuadro 9.5 que el método más eficiente es la criba de Atkin. Los otros métodos se utilizarán cuando sea conveniente para obtener una respuesta en un tiempo más eficiente. Como se trata de una prueba experimental en cada máquina los tiempos que se obtengan pueden ser diferentes pero la relación será la misma.

Algoritmo	Tiempo
Divisiones Sucesivas	13m45.480s
Divisiones Sucesivas Mejorado	6m59.346s
Criba	0m12.545s
Criba Bit Set	0m12.545s
División Solo Primos	1m53.367s
Miller Rabin	5m17.904s
Criba de Atkin	0m5.716s

Cuadro 9.5: Tiempo de proceso de los algoritmos presentados

9.12. Números Grandes

La aritmética de números grandes es esencial en varios campos tales como la criptografía. Las bibliotecas de Java incluyen una variedad de funciones para el tratamiento de éstos números. Cabe hacer notar que cuando se trata con números pequeños estas librerías son más lentas que los métodos descritos con anterioridad.

- Constructores

  BigInteger(String val) Permite construir un número grande a partir de una cadena

  BigInteger(int largo, int certeza, Random r) Permite construir un número probablemente primo de la longitud de bits especificada, la certeza especificada y random es un generador de números aleatorios.

- Métodos

  add(BigInteger val) Devuelve $this + val$

  compareTo(BigInteger val) Compara *this* con *val* y devuelve $-1, 0, 1$ para los casos que sean $<, =, >$ respectivamente

  intValue() Devuelve el valor entero de *this*

  gcd(BigInteger val) Halla el máximo común divisor entre *this* y *val*

  isProbablePrime(int certeza) .- Prueba si el número es probablemente primo con el grado de certeza especificado. Devuelve falso si el número es compuesto.

  mod(BigInteger m) Devuelve el valor *this* mod m

  modInverse(BigInteger m) Retorna el valor $this^{-1}$ mod m

  modPow(BigInteger exponente, BigInteger m) Devuelve $this^{exponente}$ mod m

  multiply(BigInteger val) Devuelve $this * val$

  pow(int exponente) Devuelve $this^{exponente}$

  probablePrime(int longitud, Random rnd) Devuelve un número probablemente primo. El grado de certeza es 100. Y *rnd* es un generador de números aleatorios.

  remainder(BigInteger val) Devuelve $this \% val$

  subtract(BigInteger val) Devuelve $this - val$

  toString(int radix) Devuelve una cadena en la base especificada.

9.12.1. Ejemplo

Hallar un número primo aleatorio de 512 Bits. La función probablePrime utiliza para ésto las pruebas de Miller-Rabin y Lucas-Lehmer.

```
/**
*Programa para hallar un numero
* probable primo de 512 bits
* @author Jorge Teran
*
*/

import java.math.BigInteger;
import java.util.Random;

public class PrimoGrande {
     public static void main(String args[]) {
         Random rnd = new Random(0);
         // generar un probable primo de 512 bits
         // con una probabilidad de que sea compuesto
         // de 1 en 2 elevado a 100.
         BigInteger prime =
         BigInteger.probablePrime(512, rnd);

         System.out.println(prime);
         }
}
```

Un ejemplo de ejecución del programa es:

```
11768683740856488545342184179370880934722814668913767795511
29768394354858651998578795085800129797731017311099676317309
35911488339878356533264880532790710 57
```

Veamos otro ejemplo. Supongamos que queremos desarrollar el algoritmo de encriptación de llave pública RSA . El algoritmo consiste en hallar dos números d, e que cumplan la propiedades siguientes: $c = m^e \ mod \ n$ y $m = c^d \ mod \ n$. Para resolver esto procedemos como sigue:

1. Escoger dos números primos p y q con $p \neq q$

   ```
   BigInteger p = new BigInteger(longitud/2, 100, r);
   BigInteger q = new BigInteger(longitud/2, 100, r);
   ```

2. Calcular $n = pq$

   ```
   n = p.multiply(q);
   ```

3. Escoger e que sea relativamnete primo a (p-1)(q-1)

   ```
   BigInteger m = (p.subtract(BigInteger.ONE))
         .multiply(q.subtract(BigInteger.ONE));
   e = new BigInteger("3");
   while(m.gcd(e).intValue() > 1)
      e = e.add(new BigInteger("2"));
   ```

4. Calcular el multiplicativo inverso de e

```
d = e.modInverse(m);
```

5. Publicar la clave pública como el par $p = (e, n)$
6. Publicar la clave privada como el par $s = (d, n)$

Veamos el código completo. Aquí hay que hacer notar que en Java existen dos clases para hallar números aleatoreos, *Random* y *SecureRandom*. La diferencia entre ambas está en que la clase *SecureRandom* hace la semilla inicial de generación de números menos predecibles. Esto significa que de una ejecución a otra la secuencia aleatoria no va a ser la misma.

```
/**
* Programa para probar el cifrado
* de clave publica
* @author Jorge Teran
* @param args ninguno
*/

import java.math.BigInteger;
import java.security.SecureRandom;

class Rsa
{
    private BigInteger n, d, e;
    public Rsa(int longitud){
        SecureRandom r = new SecureRandom();
        BigInteger p = new BigInteger(longitud/2, 100, r);
        BigInteger q = new BigInteger(longitud/2, 100, r);
        n = p.multiply(q);
        BigInteger m = (p.subtract(BigInteger.ONE))
        .multiply(q.subtract(BigInteger.ONE));
        e = new BigInteger("3");
        while(m.gcd(e).intValue() >1)
            e = e.add(new BigInteger("2"));
        d = e.modInverse(m);
    }
    public BigInteger cifrar(BigInteger mensaje)
    {
        return mensaje.modPow(e, n);
    }
    public BigInteger decifrar(BigInteger mensaje)
    {
        return mensaje.modPow(d, n);
    }
    public static void main(String[] args) {
        Rsa a = new Rsa (100); //Hallar d y e de 100 Bits
        BigInteger mensaje=new BigInteger("64656667686970");
        BigInteger cifrado= a.cifrar(mensaje);
        BigInteger limpio= a.decifrar(cifrado);
        System.out.println(mensaje);
        System.out.println(cifrado);
        System.out.println(limpio);
    }
}
```

9.13. Lecturas para Profundizar

Para profundizar el tratamiento de los números primos se puede leer Factorization and Primality Testing [Bre88], Primes and Programming [Gib93], y un texto con muchas curiosidades Prime Numbers - The Most Mysterious Figures in Math [Wel05].

En la (http://mathworld.wolfram.com/search/) enciclopedia de matemáticas Mathworld se pueden encontrar otras pruebas de primalidad tanto probabilísticas, como determinísticas de las que mencionamos: Curva elíptica, test de primalidad de Adleman-Pomerance-Rumely, AKS de Agragual, Lucas-Lehmer, Ward's y otros.

Muchos de éstos métodos no se han visto efectivos para la prueba de primalidad cuando se trata de números grandes.

9.14. Ejemplos de aplicación

Algunos problemas se pueden resolver directamente con los algoritmos mostrados, sin embargo, otros requieren una técnica un poco diferente. En algunos casos es mejor construir los posibles resultados que probar todos los números.

Analicemos el problema denominado *primos redondos*. Este problema se describe en la parte de ejercicios.

La definición indica que un número primo es *redondos* cuando al quitar sucesivamente los dígitos de derecha a izquierda los resultados que obtenemos también son números primos. Por ejemplo del número 719, quitamos el 9 y tenemos 71 que es primo. Luego quitamos el 1 obteniendo 7 que también es primo.

Para resolver este problema tenemos algunas dificultades, primero que se pide que se evalúen todos los números hasta 10^10 y en nuestra criba de Eratostenes solo se puede evaluar hasta 10^7.

En este caso utilizaremos la técnica de construir los números que se piden, en lugar de, partir de un número primo y verificar si cumple la propiedad pedida.

Los números primos terminan en $1, 3, 7, 9$ excepto el número 2. Con esta propiedad podemos generar números de la siguiente forma:

Inicialmente comenzamos con los primos de un dígito. Tomamos el número 2 y agregamos el las posibles terminaciones, para obtener $21, 23, 27, 29$. Verificamos para el primer valor si es primo o no lo es. En este caso el 23 es primo, por lo tanto repetimos el proceso para obtener $231, 233, 237, 239$. Verificamos cada uno de ellos para saber si es primo, en cuyo caso repetimos el proceso para los que sean primos.

Posteriormente seguimos con los número $2, 3, 5, 7, 9$ hasta terminar de procesar todos los números.

```
/**
* Solucion al problema numeros redondos
* son los que truncando el ultimo digito tambien
* son primos
* @author Jorge Teran
**/
import java.util.Scanner;
```

```
public class NumerosRedondos {
    static boolean esPrimo(long p){
        //System.out.println(p);

        for(int i=2;(long)(i*i)<=p;i++)
            if (p% i==0)
            return false;
        return true;
    }

    public static int n = 1000;
    public static long[] p = new long[n + 1];

    public static void main(String[] args) {
        int i = 2, j = 0, k=0,a=0, b=0,a1,b1, contar;
        //generar los posibles numeros pimos Terminan
        // en 1,3,7,9
        //los primos de un digito el uno fue incluido por def.
        p[0]=1;
        p[1]=3;
        p[2]=7;
        p[3]=9;
        p[4]=2;
        a=0; b=4; contar=4;
        for (i=0;i<9;i++){
            for(j=a;j<=b;j++){
                for (k=0;k<4;k++){
                    if (esPrimo(p[j]*10+p[k])) {
                        contar++;
                        p[contar]=p[j]*10+p[k];
                    }
                }
            }
            a=b+1;
            b=contar;
            //System.out.println(a+'' "+b);
        }
        System.out.println(p[contar]);
        contar=0;
        for (long z : p)
            if (z>0){
            //System.out.print(z+'' ");
            contar++;
        }
        System.out.println(contar);
    }
}
```

Analicemos otro ejercicio, esta vez veamos el ejercicio *casi primos* descritos en la sección de ejercicios.

Un número se denomina *casi primo* si no es primo y tiene un solo factor. Una primera aproximación puede ser tomar todos los números no primos y hallar sus factores. Contamos cuantos factores se tienen

y si es uno hacemos la cuenta. Con este algoritmo encontramos una solución muy lenta para los valores extremos.

```
/**
* Solucion lenta al problema CasiPrimo
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class CasiPrimoSlow {

    public static int n = 10000000;
    public static int[] p = new int[n + 1];

    public static void main(String[] args) {
        int contar = 0;
        int i = 2, j = 0, a, b;
        // construccion de la criba
        for (i = 2; i * i <= n; i = i + 1)
            // funciona en criba
        if (p[i] == 0) {
            for (j = i + i; j <= n; j = j + i) {
                p[j] = 1;
            }
        }
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            a = in.nextInt();
            b = in.nextInt();
            contar = 0;
            for (i = a; i <= b; i++) {
                if (p[i] != 0) {
                    if (Factores(i)) {
                        contar++;
                    }
                }
            }
            System.out.println(contar);
        }
    }

    static boolean Factores(int N) {
        int s = 0;
        for (int i = 2; i <= N / i; i++) {
            if (p[i] == 0) {
                if (N                    s++;
                }
                while (N                    N = N / i;
                }
            }
        }
        if (N != 1) {
            s ++;
        }
        return (s<2);
```

```
    }
}
```

Para resolver este ejercicio eficientemente recurriremos a una modificación en la criba de Eratostenes. Recordemos que lo que hacemos es marcar los números compuestos. Cada vez que marcamos un número es porque este tiene un factor, entonces cambiemos la lógica. Sumemos 1 en lugar de colocar en 1. Ahora la criba almacena el número de factores, con lo que la cuenta se hace trivial. El código muestra la solución.

```
/**
* Solucion final al problema Casi primos
*
* @author Jorge Teran
**/
import java.util.Scanner;

public class CasiPrimo {

    public static int n = 10000000;
    public static int[] p = new int[n + 1];

    public static void main(String[] args) {
        int contar = 0;
        int i = 2, j = 0, a, b;
        // construccion de la criba
        for (i = 2; i <= n; i++)
            if (p[i] == 0) {
            for (j = i + i; j <= n; j = j + i) {
                p[j] ++;
            }
        }
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            contar = 0;
            a = in.nextInt();
            b = in.nextInt();
            for (i = a; i <= b; i++) {
                if (p[i] == 1) {
                    contar++;
                }
            }
            System.out.println(contar);
        }
    }
}
```

Como ve una pequeña variación a la criba puede proporcionar resultados adicionales muy interesantes.

9.15. Ejercicios

1. Divisores
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Dado un entero positivo mayor que cero, calcula el número de divisores positivos que tiene. A es divisor de B si y sólo si el residuo de B entre A es cero y A es un entero.

   El tiempo de proceso para este programa es de 1 segundo.

Input

Cada caso de prueba consiste de una línea que contiene un entero N ($1 \leq N < 2^{31}$). La entrada termina con un caso N = 0, este último caso no debe producir salida alguna.

Output

Para cada caso de prueba imprime dos líneas: en la primera el número de divisores positivos que tiene N, y en la segunda una lista ordenada de menor a mayor de todos los divisores positivos de N.

Ejemplo de entrada	**Ejemplo de salida**
10 12 0	4 1 2 5 10 6 1 2 3 4 6 12

2. Divisibilidad por dígitos
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Dado un número entero diga por cuantos de sus dígitos es divisible el número. Utilice todas las reglas de divisibilidad.

   Por ejemplo el número 12345 es divisible por $1, 3$ y 5 por lo que el resultado esperado es 3.

Input

La entrada consiste de varios casos de prueba, en cada línea se tiene un número entero n, $10000 \leq n \leq 999999999$, (de 5 a 9 dígitos).

Output

Por cada línea de entrada imprima en una línea la cantidad de dígitos por los que es divisible.

Ejemplo de entrada	Ejemplo de salida
12345	3
661232	3
52527	0
730000000	0

3. Divisibilidad
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

A usted le dan un número entero, del cual debe sustituir los dos últimos dígitos de tal manera que el número resultante sea divisible por un factor dado, que se tiene como dato.

Los dos últimos números adicionados deben ser el número más pequeño posible. Por ejemplo:

- Si num = 275, y factor = 5, entonces la respuesta es "00", porque 200 es divisible por 5.
- Si num = 1021, y factor = 11, entonces la respuesta es "01", porque 1001 es divisible por 11.
- Si num = 70000, y factor = 17, entonces la respuesta es "06", porque 70.006 es divisible por 17.

Input

La entrada consiste de múltiples casos de prueba cada uno en una línea. Cada caso de prueba consiste de dos números enteros A, B que son el número y el factor respectivamente. $1 \leq A \leq 2000000000$ y $1 \leq B \leq 100$

Output

Por cada caso de prueba escriba en una línea, el número más pequeño que hay que remplazar para ser divisible.

Ejemplo de entrada	**Ejemplo de salida**
100 5	00
1000 3	02
23442 75	00
428392 17	15
32442 99	72

4. Divisores de Factorial
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   La función factorial, $n!$, se define para todos los enteros no negativos n de la siguiente manera:

$$0! = 1$$

$$n! = n \times (n - 11)! \; dado \; que \; (n > 0)$$

   El problema consiste en saber si $n!$ es divisible por un número m.

Input

La entrada consiste de varias líneas y cada una de ellas con dos enteros no negativos, n y m, ambos menores que 231 y termina cuando no hay más datos de prueba.

Output

Por cada línea de la entrada, mostrar otra que indique si m divide, o no, a $n!$, utilizando el formato que aparece en el ejemplo de salida.

Ejemplo de entrada	**Ejemplo de salida**
6 9 6 27 20 10000 20 100000 1000 1009	9 divide 6! 27 no divide 6! 10000 divide 20! 100000 no divide 20! 1009 no divide 1000!

5. Lejos de los Primos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un número primo es un entero mayor que 1 que no tiene divisores positivos otros que 1 y a si mismo. Los primeros números primos son: $2, 3, 5, 7, 11, 13, 17, \ldots$ El número N se considera *lejano de un número primo* si no existe un número primo entre $N-10$ y $N+10$, inclusive, por ejemplo todos los números N-10, N-9, ..., N-1, N, N+1, ..., N+9, N+10 no son primos.

Dado un entero A y un entero B. Devuelva el número de números *lejanos de primos* entre A y B, inclusive. Sugerencia utilice una criba para hallar la solución.

Considere los núneros $A = 3328$ y $B = 4100$, existen 4 números *lejanos de primos* que son $3480, 3750, 3978$ y 4038.

Input

La entrada consiste de varias líneas que contienen dos números enteros $0 \leq A, B \leq 100000$ y $0 \leq (B - A) \leq 1000$. La entrada termina cuando no hay más datos

Output

En la salida imprima un solo número entero indicando la cantidad de números *lejano de un número primo* que existen entre A y B.

Ejemplo de entrada	**Ejemplo de salida**
3328 4100 10 1000 19240 19710 23659 24065 97001 97691	4 0 53 20 89

6. DePrimo

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un número se denomina *deprimo* si la suma de sus factores primos es un número primo. Dados los números a, b cuente cuántos números *deprimo* existen entre a, b donde $2 < a < b \leq 5000000$.

Veamos algunos ejemplos los números $2, 3, 5, 7$ son números primos, por lo que también son deprimos.

El número 4 tiene como sus factores primos el $2, 2$ como no consideramos los factores repetidos la suma de sus factores primos es 2 por lo que es un deprimo. El número 6 tiene dos factores el 2 y el 3 la suma de sus factores $2 + 3 = 5$ que es un número primo por lo que el número 6 es un número deprimo.

Si contamos cuantos de primos existen entre 2 y 5 inclusive tenemos los siguientes $2, 3, 4, 5$ cuatro números por lo que la respuesta es 4

Input

La entrada consiste en los números a, b separados por un espacio. Termina cuando no hay más datos.

Output

La salida es un solo número que indica cuantos números *deprimo* existen entre a, b.

Ejemplo de entrada	**Ejemplo de salida**
2 5 10 21 100000 120000 2 5000000	4 9 3972 846027

7. Casi Primos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números casi primos son números no-primos que son divisibles por solo un número primo. En este problema tu trabajo es escribir un programa que encuentre la cantidad de número casi primos dentro de cierto rango.

No se consideran casi primos los números primos.

Veamos un ejemplo, si tomamos el rango entre 2 y 10 solo hay 3 números Casi primos:

- El 4 solo divisible por el 2, por lo que es *casi primo*
- El 6 es divisible por 2 y 3, por lo que **no** es *casi primo*
- El 8 solo divisible por el 2, por lo que es *casi primo*
- El 9 solo divisible por el 3, por lo que es *casi primo*
- El 10 es divisible por 2 y 5, por lo que no es *casi primo*
- Los números $2, 3, 5, 7$ no son *casi primo* son primos.

Input

La primera línea de la entrada contiene un entero N ($N \leq 600$) que indica cuantos conjuntos de datos siguen. Cada una de las siguientes N líneas son los conjuntos de entrada. Cada conjunto contiene dos número enteros low y $high$ ($0 \leq low \leq high \leq 10^7$).

Output

Por cada línea de entrada excepto la primera usted debe producir una línea de salida. Esta línea contendrá un entero, que indica cuantos números casi primos hay dentro del rango(incluyendo) $low...high$.

Ejemplo de entrada	**Ejemplo de salida**
6 2 10 10 20 2 100 2 1000000 500000 5000000 2 10000000	3 1 10 236 241 555

8. Raíz digital Prima
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

La raíz digital de un número se halla adicionando todos lo dígitos en un número. Si el número resultante tiene más de un dígito, el proceso es repetido hasta tener un dígito.

Tu trabajo en este problema es calcular una variación de la raíz digital? una raíz digital prima. El proceso de adición descrito arriba para cuando solo queda un dígito, pero podemos parar en el número original, o en cualquier número intermedio (formado por la adición) que sea número primo.

Si el proceso continúa y el resultado es un dígito que no es primo, entonces el número original no tiene raíz digital prima.

Un entero mayor que uno es llamado número primo si tiene solo dos divisores, el uno y si mismo. Por ejemplo:

- Los primeros seis primos son 2, 3, 5, 7, 11, y 13.
- El número 6 tiene cuatro divisores: 6, 3, 2, y 1. Por eso 6 no es primo.
- Advertencia: en número 1 no es primo.

Ejemplos:

- 1 Este no es un número primo, así que 1 no tiene raíz digital prima.
- 3 Este es un número primo, así que la raíz digital prima de 3 es 3.
- 4 Este no es un número primo, así que 4 no tiene raíz digital prima.
- 11 Este es un número primo, así que la raíz digital prima de 11 es 11.
- 642 Este no es un número primo, así que sumando $6 + 4 + 2 = 12$. Este no es un número primo, así que sumando $1 + 2 = 3$. Este si es un número primo, así que la raíz digital prima de 642 es 3. 128 Este no es un número primo, así que sumando $1 + 2 + 8 = 11$. Este es un número primo, así que la raíz digital prima de 128 es 11.
- 886 Este no es un número primo, así que sumando $8 + 8 + 6 = 22$. Este no es un número primo, así que sumando $2 + 2 = 4$. Este no es un número primo, así que 886 no tiene raíz digital prima.

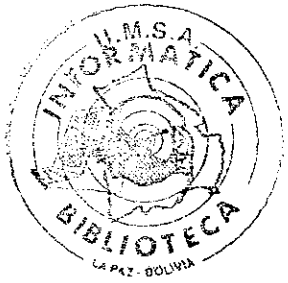
Input

La primera línea de la entrada contendrá el número de casos de prueba. Cada caso de prueba viene en una línea y contendrá un entero en cada línea en el rango de 0 a 999999 inclusive.

Output

Si el número ingresado tiene raíz digital prima, entonces se debe desplegar el valor original y el valor de la raíz digital prima, caso contrario se despliega el valor original seguido por la palabra *none*, los valores deben estar alineados con una justificación derecha de 7 espacios, como se muestra en el ejemplo de salida.

Ejemplo de entrada	Ejemplo de salida
5 134 11 642567 912367 9234567	134 none 11 11 642567 3 912367 912367 9234567 none



9. Factorizar un Número Grande
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Se tiene un número compuesto $2 \leq n \leq 2^{31}$. Hallar sus factores primos. Como ejemplo los factores primos de 36 son $2, 2, 3, 3$.

   Si el número es primo por ejemplo el número 7 se imprimirá un solo factor el 7.

Input

La entrada consiste de varios casos de prueba. En cada línea se tiene un número entero positivo compuesto. La entrada termina cuando no hay más datos.

Output

Para cada caso de prueba escriba los factores primos del número en una sola línea separados por un espacio.

Ejemplo de entrada	**Ejemplo de salida**
36	2 2 3 3
1504703107	1504703107
600851475143	71 839 1471 6857
60085147514356	2 2 83 179 683 1480319
9223372036854775805	5 23 53301701 1504703107
9223372036854775807	7 7 73 127 337 92737 649657

10. Pares de Ruth-Aaron

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Este nombre fue dado por Carl Pomerance en honor a Babe Ruth y Hank Aaron estrellas de las ligas del baisballl estadounidense. El récord que tenía Ruth era de 714 entradas, que superada por Aron el 8 de abril de 1974, con un total de 715 entradas. Uno de los estudiantes de Pomerance advirtió que la suma de los factores primos de 714 y 715 eran iguales. Veamos:

Si factorizamos ambos números obtenemos las siguientes descomposiciones:

$$714 = 2 \times 3 \times 7 \times 17$$

$$715 = 5 \times 11 \times 13$$

Si nos fijamos en las sumas de ambas factorizaciones tenemos que:

$$2 + 3 + 7 + 17 = 5 + 11 + 13 = 29$$

A los números que satisfacen esta propiedad, es decir, a los pares consecutivos cuya descomposición en factores primos tienen la misma suma, Pomerance les llamó pares de *Ruth-Aaron*. Y claro está, los ordenadores son fantásticos. Pomerance descubrió que entre los números menores a 20,000 hay 26 pares de *Ruth-Aaron*. El mayor en éste rango lo forman el 18,490 y el 18,491.

Si solo consideramos los factores primos diferentes los primeros pares de *Ruth-Aaron* son: $(5,6)$, $(24,25)$, $(49,50)$, $(77,78)$, $(104,105)$, $(153,154)$, $(369,370)$, $(492,493)$, $(714,715)$, $(1682,1683)$ y $(2107,2108)$

Si contamos los factores primos repetidos, por ejemplo $8 = 2 \times 2 \times 2$ y $9 = 3 \times 3$ con $2+2+2+ = 3+3$ los primeros pares de *Ruth-Aaron* son: $(5,6)$, $(8,9)$, $(15,16)$, $(77,78)$, $(125,126)$, $(714,715)$, $(948,949)$ y $(1330,1331)$.

La intersección de las dos listas es: $(5,6)$, $(77,78)$, $(714,715)$.

Input

La entrada consiste de varios casos de prueba. La primera línea de un caso de prueba contiene un entero N representando el número de casos de prueba ($1 \leq N \leq 100$). En las siguientes líneas de cada caso de prueba contienen dos enteros a, b ($2 \leq a, b \leq 1000000$),

Output

Para cada caso de prueba su programa debe mostrar en la salida los pares de Ruth-Aron, entre a y b, que pertenecen a la intersección de ambas listas, la que incluye factores primos repetidos y la que no incluye factores primos repetidos.

Ejemplo de entrada	**Ejemplo de salida**
1 2 1331	5 6 77 78 714 715

11. Primos de Izquierda Derecha y de Costado
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Existen varias características interesantes de algunos números primos. Consideremos por ejemplo el número 1429 tiene la propiedad que si escribimos en orden reverso 9241 también es un número primo. Si escribimos los dígitos de este número en cualquier orden, 1249, 9421, 4129 todos ellos son primos.

Input

No hay datos de entrada.

Output

Su programa debe generar todos los números primos que cumplan las características mencionadas menores a 10^8. Cada número en una sola línea. En la salida se muestra como ejemplo un solo número

Ejemplo de entrada	**Ejemplo de salida**
	1249

12. Pares de Ruth-Aaron SIN repetición
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Este nombre fue dado por Carl Pomerance en honor a Babe Ruth y Hank Aaron estrellas de las ligas del baisballl estadounidense. El récord que tenia Ruth era de 714 entradas, que superada por Aron el 8 de abril de 1974, con un total de 715 entradas. Uno de los estudiantes de Pomerance advirtió que la suma de los factores primos de 714 y 715 eran iguales. Veamos:

Si factorizamos ambos números obtenemos las siguientes descomposiciones:

$$714 = 2 \times 3 \times 7 \times 17$$

$$715 = 5 \times 11 \times 13$$

Si nos fijamos en las sumas de ambas factorizaciones tenemos que:

$$2 + 3 + 7 + 17 = 5 + 11 + 13 = 29$$

A los números que satisfacen esta propiedad, es decir, a los pares consecutivos cuya descomposición en factores primos tienen la misma suma, Pomerance les llamó pares de *Ruth-Aaron*. Y claro está, los ordenadores son fantásticos. Pomerance descubrió que entre los números menores a 20,000 hay 26 pares de *Ruth-Aaron*. El mayor en este rango lo forman el 18,490 y el 18,491.

Si solo consideramos solo los factores primos diferentes, los primeros pares de *Ruth-Aaron* son: $(5,6)$, $(24,25)$, $(49,50)$, $(77,78)$, $(104,105)$, $(153,154)$, $(369,370)$, $(492,493)$, $(714,715)$, $(1682,1683)$ y $(2107,2108)$

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba contiene dos enteros a, b $(2 \leq a, b \leq 1000000)$. La entrada termina cuando no hay más datos.

Output

Para cada caso de prueba su programa debe mostrar en la salida los pares de Ruth-Aron, entre a y b, que incluyen factores primos repetidos.

Ejemplo de entrada	Ejemplo de salida
2 2110 5000 10000 60000 70000 901000 903000	5 6 24 25 49 50 77 78 104 105 153 154 369 370 492 493 714 715 1682 1683 2107 2108 5405 5406 6556 6557 6811 6812 8855 8856 9800 9801 61760 61761 63665 63666 64232 64233 901747 901748

13. Pares de Ruth-Aaron CON Factores Repetidos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Este nombre fue dado por Carl Pomerance en honor a Babe Ruth y Hank Aaron estrellas de las ligas del baisballl estadounidense. El récord que tenía Ruth era de 714 entradas, que superada por Aron el 8 de abril de 1974, con un total de 715 entradas. Uno de los estudiantes de Pomerance advirtió que la suma de los factores primos de 714 y 715 eran iguales. Veamos:

Si factorizamos ambos números obtenemos las siguientes descomposiciones:

$$714 = 2 \times 3 \times 7 \times 17$$

$$715 = 5 \times 11 \times 13$$

Si nos fijamos en las sumas de ambas factorizaciones tenemos que:

$$2 + 3 + 7 + 17 = 5 + 11 + 13 = 29$$

A los números que satisfacen esta propiedad, es decir, a los pares consecutivos cuya descomposición en factores primos tienen la misma suma, Pomerance les llamó pares de *Ruth-Aaron*. Y claro está, los ordenadores son fantásticos. Pomerance descubrió que entre los números menores a 20,000 hay 26 pares de *Ruth-Aaron*. El mayor en este rango lo forman el 18,490 y el 18,491.

Si contamos los factores primos repetidos, por ejemplo $8 = 2 \times 2 \times 2$ y $9 = 3 \times 3$ con $2+2+2+ = 3+3$ los primeros pares de *Ruth-Aaron* son: $(5, 6)$, $(8, 9)$, $(15, 16)$, $(77, 78)$, $(125, 126)$, $(714, 715)$, $(948, 949)$ y $(1330, 1331)$.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba contiene dos enteros a, b $(2 \leq a, b \leq 1000000)$,

Output

Para cada caso de prueba su programa debe mostrar en la salida los pares de Ruth-Aron, entre a y b, que incluye factores primos repetidos. La entrada termina cuando no hay más datos.

Ejemplo de entrada	**Ejemplo de salida**
2 2110 5000 10000 60000 70000 901000 903000	5 6 8 9 15 16 77 78 125 126 714 715 948 949 1330 1331 1520 1521 1862 1863 5405 5406 5560 5561 5959 5960 6867 6868 8280 8281 8463 8464 63344 63345 63426 63427 68264 68265 68949 68950

14. Contando Primos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números primos son aquellos que son solo divisibles por si mismo o el uno. La lista de los primeros números primos es $2, 3, 5, 7, 9, 11, 13, 17, 19$

Dados $0 \leq a, b \leq 10^7$ contar cuantos primos hay en ese rango.

Input

La entrada consiste de varios casos de prueba. La primera línea contiene el número de casos de prueba. Las líneas siguientes corresponden a los casos de prueba, cada caso de prueba contiene dos enteros a, b, La entrada termina cuando no hay más datos.

Output

Para cada caso de prueba su programa debe mostrar, en la salida, una línea con el número de primos existente entre a y b inclusive.

Ejemplo de entrada	**Ejemplo de salida**
4 2 1000 100000 1000000 1000000 10000000 2 10000000	168 68906 586081 664579

15. Primos Truncables

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

El número $n = 3797$ tiene una propiedad muy interesante. Siendo n un número primo es posible quitar continuamente sus dígitos de izquierda a derecha en forma continúa y cada uno de los números remanentes también es primo. En cada etapa : 3797, 797, 97, y 7 también son números primos. Similarmente si trabajamos de derecha a izquierda: 3797, 379, 37,y 3.

Tome en cuenta que los números $2, 3, 5$ y 7 no se consideran truncables. Note que el número 1 no es primo.

Dados $0 \leq a, b \leq 10^7$ listar los primos truncables que hay en ese rango.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba contiene dos enteros a, b $(0 \leq a, b \leq 10^7)$, La entrada termina cuando no hay mas datos.

Output

Para cada caso de prueba su programa debe listar en una línea, los primos truncables que hay en el rango a y b inclusive. Si no hay ningún primo truncable imprima una línea en blanco. La entrada termina cuando no hay mas datos.

Ejemplo de entrada	**Ejemplo de salida**
0 50 3700 3800	23 31 37 3797

16. Contando Primos Gemelos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números primos son aquellos que son solo divisibles por si mismo o el uno. La lista de los primeros números primos es $2, 3, 5, 7, 9, 11, 13, 17, 19$

Los números primos gemelos son aquello que al sumarle 2 también son primos. Los primeros números primos gemelos son: $(3, 5)$, $(5, 7)$, $(11, 13)$, $(17, 19)$, $(29, 31)$, $(41, 43)$.

Todos los números primos gemelos tienen la forma $6n \pm 1$

Dados $0 \leq a, b \leq 10^7$, contar cuántos primos gemelos hay en ese rango.

Input

La entrada consiste de varios casos de prueba. La primera línea contiene el número de casos de prueba. Las líneas siguientes corresponden a los casos de prueba, cada caso de prueba contiene dos enteros a, b, La entrada termina cuando no hay mas datos.

Output

Para cada caso de prueba su programa debe mostrar, en la salida, una línea con el número de primos gemelos existente entre a y b inclusive.

Ejemplo de entrada	**Ejemplo de salida**
6 2 50 2 500 2 5000 2 50000 2 500000 2 5000000	12 48 252 1410 9130 64926

17. Primos Redondos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

El número $n = 719$ tiene una propiedad muy interesante. Siendo n un número primo es posible quitar continuamente sus dígitos de derecha a izquierda en forma continua y cada uno de los números remanentes también es primo. En cada etapa : 719, 71, y 7 también son números primos.

Tome en cuenta que los números $1, 2, 3, 5$ y el 7 se consideran *redondos*. Note que el número 1 no es primo, pero en este ejercicio se lo considerar como tal. Por este motivo el 11 se considera un número redondo.

El tiempo de proceso para este programa es de 1 segundo.

Para resolver el problema en un tiempo adecuado tome en cuenta que los números primos terminan en 1,3,7 y 9. Con esta idea construya todos los números que cumplan la definición dada.

Input

No hay datos de entrada

Output

Su programa de contar cuanto números *primos redondos* existen entre 1 y 10^{10}

Ejemplo de entrada	**Ejemplo de salida**
	140

18. Criba de Eratóstenes
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

La Criba de Eratóstenes es un método muy antiguo para hallar todos los números primos entre un número y un máximo dado.

Funciona como sigue:

Construya una lista de todos los números entre 2 y N inclusive:

```
for x = 2 to N
   if (x == no marcado)
      for y = 2 to N/x
         if (x*y == no marcado) marcar x*y
      end for
   end if
end for
```

De esta manera todos los números primos **no** estarán marcados y los números no primos estarán marcados.

Lo que se quiere es imprimir el último número marcado.

Por ejemplo: si la entrada es 18 la respuesta es 15. Los números marcados fueron:

- Cuando $x = 2$ se marcaron los números $4, 6, 8, 10, 12, 14, 16, 18$
- Cuando $x = 3$ se marcaron los números $9, 15$, otros múltiplos fueron marcados con anterioridad.
- Los números $5, 7, 11, 13, 17$ no fueron marcados.

El último número marcado fue el 15.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba contiene un número $4 \leq N \leq 2000000000$, (2^{109}). La entrada termina cuando no hay más datos de prueba.

Output

En la salida se debe imprimir, por cada caso de prueba, el último número marcado en una sola línea.

Ejemplo de entrada	**Ejemplo de salida**
18	15
5	4
100	91
400	361

19. Números Feos

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números feos son números cuyos únicos factores primos son 2, 3, o 5.

La secuencia:
$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...$
Muestra los primeros 11 números feos. Por convención se incluye el 1.

Esto equivale a encontrar los números que se pueden formar por $2^a 3^b 5^c$. Escriba un programa que encuentre e imprima el número 1500.

Input

No existen datos de entrada en este problema.

Output

La salida debe consistir de una línea reemplazando *número* con el número calculado.

Ejemplo de entrada

```
No existen datos de entrada en este problema.
```

Ejemplo de salida

```
El numero feo 1500 es ......
```

20. Números de Carmichael
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Ciertos algoritmos criptográficos hacen uso de grandes números primos. Sin embargo, comprobar si un número muy grande es primo no resulta sencillo.

Existen pruebas probabilísticas de primalidad que ofrecen un alto grado de fiabilidad, como la prueba de Fermat. Supongamos que a es un número aleatorio entre 2 y n, donde n es el número cuya primalidad debemos comprobar. Entonces, n es *probablemente primo* si se cumple la siguiente ecuación:

$$a^n \mod n = a$$

Si un número pasa varias veces la prueba de Fermat, tiene una probabilidad muy alta de ser un número primo.

Por desgracia, no todo son buenas noticias. Algunos números compuestos (no primos) cumplen la prueba de Fermat con cualquier número inferior a ellos. Estos son los conocidos como números de Carmichael.

La tarea consiste en escribir un programa que determine si un entero dado es un número de Carmichael.

Input

La entrada consta de una serie de líneas, conteniendo cada una de ellas un entero positivo pequeño ($2 \leq n \leq 65{,}000$). La entrada finaliza cuando n = 0, valor que no debe ser procesado.

Output

Por cada número de la entrada, imprimir un mensaje que indique si se trata o no de un número de Carmichael, utilizando para ello el formato que se presenta en el ejemplo de salida.

Ejemplo de entrada	Ejemplo de salida
1729	1729 es numero de Carmichael.
17	17 es normal.
561	561 es numero de Carmichael.
1109	1109 es normal.
431	431 es normal.
0	0

21. Los números de Smith

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En 1982 el matemático Albert Wilansky descubrió mientras consultaba la guía telefónica, que el número de su cuñado H. Smith tenía la siguiente propiedad: la suma de los dígitos de ese número era igual a la suma de los dígitos de los factores primos del mismo. ¿Entendido? El teléfono de Smith era 493-7775. Este número puede expresarse como el producto de sus factores primos de la siguiente manera: $4937775 = 3 \cdot 5 \cdot 5 \cdot 65837$

La suma de todos los dígitos del número de teléfono es $4+9+3+7+7+7+5 = 42$, y la suma de los dígitos de sus factores primos es igualmente $3+5+5+6+5+8+3+7 = 42$. Wilansky bautizó este tipo de números en honor a su cuñado: los números de Smith.

Como esta propiedad es cierta para todos los números primos, Wilansky excluye éstos de la definición. Otros números de Smith son 6,036 y 9,985.

Wilansky no fue capaz de encontrar un número de Smith de mayor longitud que el del número de teléfono de su cuñado. ¿Podemos ayudarle?

Input

La entrada consta de varios casos de prueba y se indica con el número en la primera línea. Cada caso de prueba contiene un único entero positivo menor que 10^9.

Output

Por cada valor n de la entrada, se debe calcular el número de Smith más pequeño posible, siempre que sea mayor que n, y mostrarlo en una línea. Podemos asumir que dicho número existe.

Ejemplo de entrada	**Ejemplo de salida**
1 4937774	4937775

22. Contando Primos Pitagóricos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Recordando el teorema de Pitágoras, la hipotenusa C de un triángulo rectángulo, se calcula como la suma elevada al cuadrado de sus catetos. Dados los catetos A, B, $A^2 + B^2 = C^2$.

Los números primos Pitagóricos son aquellos que se obtienen de la suma de dos números elevados al cuadrado. Por ejemplo:

- $1^2 + 1^2 = 5$
- $2^2 + 1^2 = 5$
- $2^2 + 3^2 = 13$

Los primeros primos Pitagóricos son: $5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97, 101, 109, 113$.

Una propiedad que deben cumplir estos números es que son de la forma $4n + 1$, con excepción del numero 2. Por ejemplo:

- $4x1 + 1 = 5$
- $4x3 + 1 = 13$
- $4x4 + 1 = 17$

Input

La entrada consiste de varios casos de prueba. La primera línea contiene el número de casos de prueba. Las líneas siguientes corresponden a los casos de prueba, cada caso de prueba contiene dos enteros a, b.

Output

Para cada caso de prueba su programa debe mostrar, en la salida, una línea con el número de primos Pitagóricos existente entre a y b inclusive.

Ejemplo de entrada	**Ejemplo de salida**
6 2 30 2 100 2 500	4 11 44

Capítulo 10

Números de Fibonacci

10.1. Introducción

Fibonacci fue uno del los más reconocidos matemáticos de la edad media. Su nombre completo fue Leonardo de Pisa. Nació en Pisa, Italia, un importante pueblo comercial en esa época en 1175. Su padre se llamaba Guglielmo Bonacci, y como se utilizaba *fi* para decir hijo de, quedo el nombre de Fibonacci.

Leonardo viajó extensivamente por la costa del mediterráneo donde aprendió la forma en la que los comerciantes hindúes y árabes utilizaban la aritmética. Popularizó los conocimientos de los árabes introduciendo los números arábicos al mundo occidental.

La serie $1, 1, 2, 3, 5, 8....$ lo hizo famoso y el nombre de la serie lleva su nombre. El origen de estos números se dice que se originó cuando cierto hombre que tenía una pareja de conejos en un lugar cerrado y deseaba saber cuántas parejas de conejos tendrá a partir de este par en un año. Para calcular este valor llamó a Fibonacci que resolvió el problema con el siguiente análisis. Si cada pareja tiene una pareja de conejos cada mes, y en el segundo mes los recién nacidos también van ha dar otra pareja. La siguiente tabla muestra como va creciendo la población de conejos:

Mes	Conejos nacidos	Parejas Totales
Fin del mes 0	0 conejos	0
Comienzo de mes 1	Nace una pareja	1
Fin del mes 1	La pareja tiene un mes de edad y se cruza	1
Fin del mes 2	La pareja que ya tiene un mes se vuelve a cruzar tenemos nace 1 par de conejos	$1 + 1 = 2$
Fin del mes 3	Nacen los 2 pares de parejas de conejos que cruzaron el mes anterior y se vuelven a cruzar	$2 + 1 = 3$
Fin del mes 4	Se cruzan los 3 pares de conejos conejos y nacen los 2 que cruzaron el mes anterior	$3 + 2 = 5$

Los números obtenidos son: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$ la secuencia obtenida se denomina secuencia de Fibonacci. El número siguiente se obtiene sumando los dos números anteriores. Matemáticamente $f_n = f_{n-1} + f_{n-2}$. Estas ecuaciones se denominan ecuaciones recurrentes o de recurrencia.

10.2. Programando la secuencia

Para hallar esta secuencia inicialmente la programemos utilizando la fórmula, sumar un valor con los dos anteriores, Primero mostramos el primer valor de la serie, luego iteramos hasta hallar los valores que deseamos. El programa resultante:

```
import java.util.*;
public class Fib {

    public static void main(String[] args) {
        int y=1;
        int n=0;
        int x=0;
        int fib=0;
        while (n<6){
            fib=x+y;
            System.out.print(fib+ '' ");
            x=y;
            y=fib;
            n++;
        }
    }
}
```

Como hallar la solución de la ecuación $f_n = f_{n-1} + f_{n-2}$ puede ver en [Pom93]. No es intención de este texto trabajar en matemática discreta, por lo que solo presentamos la respuesta:

$$f(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

El programa

```
import java.util.*;
public class Fib {

    public static void main(String[] args) {
        double n=6;
        double r5 = Math.sqrt(5.0);
        double fib = (Math.pow((1+r5)/2,n)-Math.pow((1-r5)/2,n))/r5;
        System.out.println(fib);
    }
}
```

Resuelve el problema de hallar la secuencia, claro está, que con un error de precisión. Para el ejemplo del programa el Fibonacci 6 da 8,000000000000002.

También es posible hallar el siguiente Fibonacci conociendo el anterior, es decir hallar f_{n+1} conociendo f_n es posible utilizando la fórmula:

$$f_{n+1} = \lfloor x + \frac{1\sqrt{5x^2})}{2} \rfloor$$

Si dividimos los valores consecutivos de la serie, cada uno con el elemento anterior, obtenemos

1/1	1
2/1	2
3/2	1.5
5/3	1.66
8/5	1.6
13/8	1.625
21/13	1.618

Si continuamos son la serie, veremos que el cociente converge a 1,6180 este valor se denomina número aúreo o de oro porque aparece en muchos aspectos de la naturaleza. Este número es:

$$f(n) = \frac{1+\sqrt{5}}{2})$$

Otra forma más eficiente de hallar los números de la secuencia de Fibonacci es utilizando matrices. Escribamos el parte del programa que hicimos para hallar la secuencia

```
while (n<6){
fib=x+y;
System.out.print(fib+ " ");
x=y;
y=fib;
n++;
```

En este código haremos los siguientes cambios:, como $fib = x + y$ remplazamos en $y = fib$, y nos queda:

```
x=y;
y=x+y;
```

Estas ecuaciones las podemos escribir en forma matricial:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{10.2.1}$$

Denominando estas matrices A, B tenemos

$$A = \begin{pmatrix} x \\ y \end{pmatrix} \tag{10.2.2}$$

$$B = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{10.2.3}$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \tag{10.2.4}$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}^4 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix} \tag{10.2.5}$$

$$\begin{pmatrix} a & b \\ b & a+b \end{pmatrix} \tag{10.2.6}$$

Calculando $B \cdot B$ se tiene que:

$$a = a \cdot a + b \cdot by$$

$$b = b \cdot a + a \cdot b + b \cdot b.$$

Calculando $A \cdot B$ se tiene que:

$$x = a \cdot x + b \cdot y$$

$$y = b \cdot x + a \cdot y + b \cdot y.$$

Programando esta solución tenemos.

```
import java.util.*;
public class Fib {
    public static void main(String[] args){
        int n=12,a=0,b=1,x=0,y=1,temp=0;
        while(n!=0){
            if(n                temp = a*a + b*b;
                b = b*a + a*b + b*b;
                a=temp;
                n=n/2;
                }
            else {
                temp = a*x + b*y;
                y = b*x + a*y + b*y;
                x=temp;
                n=n-1;
                }
            }
        System.out.println(``x= "+x);
        }
}
```

10.3. Fibonacci y el triángulo de Pascal

El triángulo de Pascal se construye colocando un 1 arriba y dos números por debajo formando un triángulo. Cada número es la suma de los dos números que tiene arriba.

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
```

Este triángulo tiene varias propiedades interesantes, en la primera de las diagonales los valores son todos 1. En la segunda diagonal los valores son la secuencia $1, 2, 3, 4, 5....$ Una característica es que es simétrico con relación al centro.

La suma de las filas tiene la propiedad de ser las potencias de 2.

$$
\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & suma = 2^0 = 1 \\
 & & & 1 & & 1 & & & & & suma = 2^1 = 2 \\
 & & 1 & & 2 & & 1 & & & & suma = 2^2 = 4 \\
 & 1 & & 3 & & 3 & & 1 & & & suma = 2^3 = 8 \\
1 & & 4 & & 6 & & 4 & & 1 & & suma = 2^4 = 16 \\
1 & & 5 & & 10 & & 10 & & 5 & 1 & suma = 2^5 = 32
\end{array}
$$

Para mostrar la relación que tiene el triángulo de Pascal con los números de Fibonacci escribamos el mismo triángulo con todos los números alineados a la izquierda:

$$
\begin{array}{cccccc}
1 & & & & & \\
1 & 1 & & & & \\
1 & 2 & 1 & & & \\
1 & 3 & 3 & 1 & & \\
1 & 4 & 6 & 4 & 1 & \\
1 & 5 & 10 & 10 & 5 & 1
\end{array}
$$

Si sumamos la primera diagonal tenemos 1. La segunda diagonal también es 1. La tercera diagonal da $1+1=2$. La cuarta $1+3+1=5$. Otra vez la secuencia de Fibonacci. Esto se puede ver que se da debido a que cada número es la suma de dos números anteriores.

10.4. Propiedades

Presentamos algunas propiedades de esta serie:

1. Hay muchas formas en las que aparecen los números de Fibonacci, por ejemplo en combinatoria, sea N tomado de K:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

$$
\begin{aligned}
\binom{0}{0} &= 1 \\
\binom{1}{0} &= 1 \\
\binom{2}{0} + \binom{2}{1} &= 2 \\
\binom{3}{0} + \binom{3}{1} &= 3 \\
\binom{4}{0} + \binom{4}{1} + \binom{3}{2} &= 5 \\
\binom{5}{0} + \binom{5}{1} + \binom{4}{2} &= 8 \\
\binom{6}{0} + \binom{6}{1} + \binom{5}{2} + \binom{4}{3} &= 13 \\
\binom{7}{0} + \binom{7}{1} + \binom{6}{2} + \binom{5}{3} &= 21
\end{aligned}
$$

2. Suma de los términos $f_1 + f_2 + f_3 + f_4 \ldots\ldots + f_n = f_{n+2} + 1$

3. Suma de los términos impares $f_1 + f_3 + f_3 + f_5 \ldots\ldots + f_{2n-1} = f_{2n}$.

4. Suma de los términos pares $f_2 + f_4 + f_6 + f_6 \ldots\ldots + f_{2n} = f_{2n-1} + 1$
5. Suma de los cuadrados $f_1^2 + f_2^2 + f_3^2 + f_4^2 \ldots\ldots + f_n^2 = f_n f_{n+1}$
6. Diferencia de cuadrados $f_{n+1}^2 - f_{n-1}^2 = f_{2n}$
7. Dados dos números Fibonacci f_a, f_b el máximo común divisor de ambos números es el número Fibonacci $f_{mcd(f_a,f_b)}$. Por ejemplo $f_8 = 21, f_{12} = 144$ entonces el $mcd(8,12) = 4$. El $mcd(21,144) = 3$ y el $f_3 = 4$
8. Si a es divisible por b entonces f_a es divisible por f_b.
9. El número f_a es par si y solo si a es múltiplo de 3.

Otras variantes de la serie de Fibonacci se obtienen cambiando los valores iniciales, la cantidad de términos a sumar, etc. Para comprender mejor esta serie se recomienda hacer los siguientes programas:

1. Escribir un programa para verificar las propiedades mostradas.
2. Escribir un programa para generar la serie $2, 5, 7, 12, 19$
3. Escribir un programa para generar la serie $1, 3, 4, 7, 11, 18$
4. Escribir un programa para generar la serie $-1, -5, -6, -11, -17$

Para más información sobre esta secuencia puede leer el libro [Fis10] y la página web [Kno10].

10.5. Ejercicios

1. Contando Fibonacci
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Dados dos números a, b donde $a < b < 2^{64}$. Contar cuantos números de Fibonacci existen que cumplen la condición $a \leq f_i \leq b$. Se le pide imprimir en la salida un número por cada caso de entrada.

Input

La entrada consiste de varias líneas. En cada línea hay dos números a, b separados por un espacio. La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada en la salida se imprimirá un número entero, que indica la cantidad de números de Fibonacci que existen en el rango.

Ejemplo de entrada	**Ejemplo de salida**
10 100 1234567890 9876543210	5 4

2. Sumando Fibonacci
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Los números de Fibonacci tienen muchas propiedades más allá de contar conejos. Podemos representar cada número entero como una suma de números Fibonacci. Por ejemplo el número 33 puede expresarse como:

$$33 = 1 * 1 + 0 * 2 + 1 * 3 + 0 * 5 + 1 * 8 + 0 * 13 + 1 * 21$$

Tomando solamente los números binarios tendríamos

$$Fib(33) = 1010101$$

Otros ejemplos son

$$Fib(18) = 000101$$

$$Fib(6) = 1001$$

Input

La entrada de datos consiste en una línea con la representación de un número como la suma de números Fibonacci. Termina cuando no hay mas datos. Vea que cuando lea la línea de entrada leerá caracteres 1 y 0 que se representan como los ascii 48 y 49 respectivamente.

Output

La salida consiste en el número entero representado por la representación leída. Se imprime un número en una línea por cada línea de entrada.

Ejemplo de entrada	**Ejemplo de salida**
1010101 000101 1001	33 18 6

3. Hallando números Fibonacci
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Los números de Fibonacci se calculan con la fórmula $f_n = f_{n-1} + f_{n-2}$

   Su tarea es la de imprimir el número Fibonacci correspondiente a un número de entrada. Por ejemplo el tercer Fibonacci es el 2. Para cada número en la entrada imprima el número Fibonacci correspondiente.

Input

La entrada consiste de números $a < 500$ que representa el número de Fibonaci que queremos hallar. Cada número está en una línea. La entrada termina cuando no hay más datos en la entrada.

Output

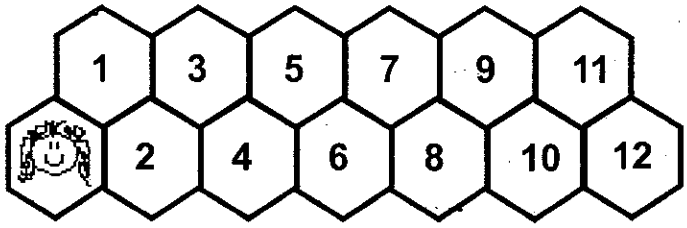
Por cada línea de entrada debe imprimir el número Fibonacci correspondiente en una línea.

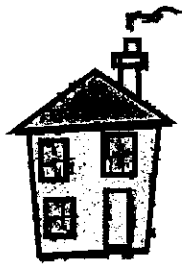
Ejemplo de entrada	**Ejemplo de salida**
4 7 8 10	3 13 21 65

4. Losas Hexagonales
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Un camino conduce a la escuela de María. Se está pavimentando con losas hexagonales en una línea recta como se muestra en la imagen. María ama las matemáticas. Cuando va a la escuela, da pasos sobre las baldosas de ese camino con estas reglas:

- Siempre se inicia desde el azulejo con la cara sonriente (Siempre es bueno comenzar con una sonrisa). Esta cerámica está siempre presente en EL INICIO de la ruta.
- Las otras losas se numeran en orden ascendente a partir del 1 de 1.
- No se le permite regresar, es decir que no debe pisar una losa que lleva un número menor que en el que se encuentra. Por seguridad Siempre da pasos de una losa a la vecina.
- El juego siempre termina en la baldosa con el número más alto.
- Cuando terminan las clases ella está tan cansada que evita el camino y camina sobre el césped.
- María no quiere repetir las secuencias de pasos sobre las losas y que le gustaría saber, si el camino es pavimentado con losas numeradas y una baldosa con la cara, ¿Cuántos días se tardará en hacer cada posible secuencia de una vez.





Input

La entrada contiene una línea por caso de prueba con el número n de losas $1 < n < 50$. La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada escriba en la salida una línea es el número de secuencias.

Ejemplo de entrada	Ejemplo de salida
1 4 2 10	1 5 2 89

5. Suma de dígitos Fibonacci
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En la lista siguiente vemos que el número Fibonacci $F(10)$ es 55

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...$$

ahora bien, si sumamos sus dígitos $5 + 5$ el resultado es 10

En el siguiente ejercicio estamos interesados en conocer para que números de Fibonacci la suma de sus dígitos representa el mismo numero de Fibonacci.

Por ejemplo $F(11) = 89$ entonces $8+9 = 17$ no suma 11 por lo que no cumple con esta propiedad.

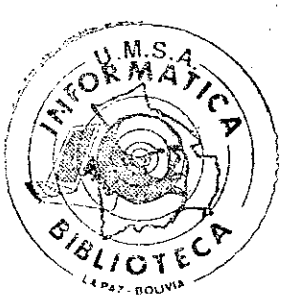
Input

No hay datos de entrada.

Output

Escriba los primeros 20 números de Fibonacci, uno en cada línea, que cumple la propiedad descrita. El ejemplo muestra los primeros 7 números.

Ejemplo de entrada	**Ejemplo de salida**
	1 1 2 3 5 8 55



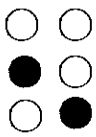
6. Acomodando clientes

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

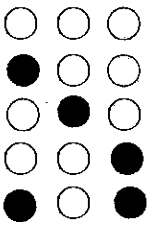
Usted tiene que acomodar a muchas personas que han venido al cine y que no pueden sentarse uno a lado del otro debido a que son muy poco amigables.

En el cine hay diferentes tipos de filas, existen filas de asientos, tres asientos, cuatro asientos, y así sucesivamente.

Por ejemplo si tiene una fila de dos asientos puede acomodar cero o una personas y existen tres formas en las que puede acomodar:



Si tenemos tres asientos en una fila podemos acomodar dos por personas como sigue:



Input

La entrada consiste de varios casos de prueba. En cada caso de prueba se tienen dos líneas. La primera línea indica el número de filas $2 < N < 50$ que tiene el cine. La segunda línea del caso de prueba tiene $2 \leq N \leq 20$ valores que representan la cantidad de asientos que tiene cada fila. La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba escriba de cuantas formas se pueden acomodar las personas en el cine.

Ejemplo de entrada	**Ejemplo de salida**
1 2 2 3 1 3	3 8 5

7. Patrones de Fibonacci
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Veamos el último dígito de cada número de Fibonacci.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...$$

La pregunta que nos hacemos es si hay un patrón para estos dígitos

$$0, 1, 1, 2, 3, 5, 8, 3, 1, 4, 5, 9, 4, 3, 7, 0, 7, ...$$

La respuesta es si, después de 60 números de vuelven a repetir, en forma cíclica.

En un caso más general podemos hallar cada uno de los números módulo $2, 3, 4...$

Por ejemplo si hallamos los valores después de hallar al módulo 2 vemos que los restos son $0, 1, 1, 0, 1, 1, 0$. La respuesta es que cada 3 números se vuelve a repetir la serie.

Input

La entrada consiste de varias líneas. En cada línea hay un número $1 \leq a \leq 100$ que utilizaremos para hallar el módulo. La entrada termina cuando no hay más datos.

Output

Por cada línea de entrada en la salida se imprimirá un número entero que indica cada cuantos números se repite la secuencia resultante de hallar el módulo a sobre los números de Fibonacci.

Ejemplo de entrada	**Ejemplo de salida**
2	3
3	8
4	6
5	20
10	60
100	1500

Capítulo 11

Algoritmos de búsqueda y clasificación

11.1. Introducción

La búsqueda y la clasificación tienen una serie de aplicaciones que serán descritas en las siguientes secciones. Se procederá, de métodos generales a casos particulares que reducen el tiempo de proceso. El proceso de especificación se realiza como se mostró en el capítulo anterior especificando las invariantes, pre y post condiciones sin entrar en la demostración de éstas. Los libros de Horstman [CSH05] proporcionan una buena descripción de las librerías de Java. Parte de este texto proviene del libro del mismo autor [Pom93] donde se desarrolla ésta temática con una profundidad diferente.

Para comprender como se derivan los algoritmos presentado se hace necesario describir los siguientes conceptos:

Precondición Son lo valores iniciales que toman las variables del programa, o también los pre requisitos necesarios para su correcto funcionamiento. Por ejemplo puede ser que una variable no sea nula, que los datos esten ordenados, la existencia de una clase, etc.

Post condición Son los valores finales que toman las variables del programa

Todo programa consiste de tres partes:

1. Inicialización
2. Preservación de propiedades
3. Terminación

La preservación de propiedades se denomina invariante y es lo que nos permite verificar y desarrollar el código.

No es intención de este texto profundizar este tema, sin embargo, es una forma en la que introduciremos los algoritmos para ordenar y buscar.

Cada ciclo en un programa tiene una propiedad invariante, esto significa que la propiedad que definamos al principio del bucle se mantendrá al final del mismo. Un propiedad puede ser una expresión matemática, una relación, una forma ú otra propiedad que se mantiene constante. Por ejemplo para

ordenar usaremos la propiedad es mayor a los elementos ya ordenados. La idea quedará más clara a medida que se expongan los algoritmos que se exponen en el capítulo.

11.2. Algoritmos de búsqueda

La búsqueda de elementos ha sido analizada en diferentes entornos, memoria, bases de datos, texto plano y otros. Cada algoritmo de búsqueda da como resultado una eficiencia diferente en función de como se ha organizado la información. La búsqueda en forma genérica se presenta como, el mecánismo de hallar un elemento en un vector del cual solo conocemos, el número de elementos que contiene. Este algoritmo, denominado búsqueda secuencial, viene especificado como sigue:

```
public class Sec {
    /* Programa de busqueda secuencial
    *@ author Jorge Teran
    */
    static int buscar(int[] v, int t) {
       for (int i = 0; i <v.length; i++)
           if (v[i] == t) {
           hallo = i;
           break;
           }
       return (hallo);
       }
```

Este algoritmo realiza como máximo n comparaciones por lo que se denomina algoritmo lineal, y es muy lento.

Es necesario conocer más sobre el conjunto de datos para mejorar el tiempo de búsqueda. Este concepto hace que se puedan realizar búsquedas específicas para cada conjunto de datos a fin de mejorar el tiempo de proceso.

Supóngase que se tiene una lista de números enteros entre 0 y n. Estos números corresponden al código de producto en un almacén. Dado un número se quiere conocer si este pertenece al almacén. Para esto podemos organizar los números de varia formas. Se puede colocar la información en un vector y proceder con el esquema de búsqueda anterior realizando n comparaciones como máximo. ¿Será posible decidir si el producto existe en el almacén con una comparación ?.

La respuesta es sí. Consideremos un código n pequeño entonces podemos armar un vector de bits en memoria, que indique que números existen y cuales no. Esto reduce el numero de comparaciones a 1.

Se quiere cambiar la búsqueda secuencial aprovechando algunas características particulares al problema, suponiendo que los datos están previamente ordenados.

Esta solución debe funcionar tanto para enteros, cadenas y reales siempre y cuando todos los elementos sean del mismo tipo. La respuesta se almacena en un entero p que representa la posición en el arreglo donde se encuentra el elemento t que estamos buscando, -1 indica que el elemento buscado no existe en el arreglo.

Esta búsqueda denominada búsqueda binaria, resuelve el problema recordando permanentemente el rango en el cual el arreglo almacena t. Inicialmente el rango es todo el arreglo y luego es reducido comparando t con el valor del medio y descartando una mitad. El proceso termina cuando se halla el valor de t o el rango es vacío. En una tabla de n elementos en el peor caso realiza $\log_2 n$ comparaciones.

La idea principal es que t debe estar en el rango del vector. Se utiliza la descripción para construir

el programa.

```
precondición el vector esta ordenado
inicializar el rango entre $0..n-1$
loop
    {invariante: el valor a buscar debe estar en el rango}
    si el rango es vacio
        terminar y avisar que t no esta en el arreglo
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

La parte esencial del programa es la invariante que al principio y final de cada iteración nos permite tener el estado del programa y formalizar la noción intuitiva que teníamos.

Se construye el programa utilizando refinamiento sucesivo haciendo que todos los pasos respeten la invariante del mismo.

Primero buscaremos una representación del rango digamos $l..u$ entonces la invariante es *el rango debe estar entre* $l..u$. El próximo paso es la inicialización y debemos estar seguros que respeten la invariante, la elección obvia es $l = 0, u = n - 1$.

Codificando un programa con los conceptos detallados previamente tenemos:

```
precondición x debe estar ordenado
post condición p especifica la posición
              o p es -1 cuando no existe
l=0; u=n-1
loop
    {invariante: debe estar en el rango l,u}
si el rango es vacio
    terminar y avisar que t no esta en el arreglo
calcular m que es el medio del rango
use m con una prueba para reducir el rango
si se encuentra t en el proceso de reducción
    terminar y comunicar su posición
```

Continuando con el programa vemos que el rango es vacío cuando $l > u$. en esta situación terminamos y devolvemos $p = -1$. Llevando esto al programa tenemos:

```
precondición x debe estar ordenado
    post condición p especifica la posición
                  o p es -1 cuando no existe
    l=0; u=n-1
    loop
        {invariante: debe estar en el rango l,u}
        if l > u
        p=-1; break;
    calcular m que es el medio del rango
    use m con una prueba para reducir el rango
    si se encuentra t en el proceso de reducción
        terminar y comunicar su posición
```

Ahora calculamos m con $m = (l + u)/2$ donde / implementa la división entera. Las siguientes líneas implican el comparar t con $x[m]$ en la que hay tres posibilidades, por igual, menor y mayor. Con lo que el programa queda como sigue:

```
precondición x debe estar ordenado
post condición p especifica la posición
               o p es -1 cuando no existe
l=0; u=n-1
loop
    {invariante: debe estar en el rango l,u}
    if l > u
    p=-1; break;
    m=(l+u)/2;
    case
        x[m]< t : l=m+1;
        x[m]= t : p=m: break;
        x[m]> t : u=m-1
```

Este algoritmo se encuentra implementado en el método *binarySearch* de *arrays* .

11.3. Clasificación

La clasificación o procedimiento de ordenar es uno de los problemas más importantes en la computación y existen un sin número de problemas de programación que requieren de ella. A continuación se detallan una serie de aplicaciones sin pretender que sean todas.

- Cómo podemos probar que en un grupo de elementos no existen elementos duplicados? Bien para esto ordenamos el conjunto y posteriormente verificamos que no existen valores tales que $x[i] = x[i + 1]$ recorriendo el conjunto para todos los valores de i permitidos.
- Cómo podemos eliminar los duplicados? Ordenamos el conjunto de datos y luego utilizamos dos índices haciendo $x[j] = x[i]$. Se incrementa el valor de i y j en cada iteración. Cuando se halla un duplicado solo incrementa i. Al finalizar ajustamos el tamaño del conjunto al valor de j.
- Supongamos que tenemos una lista de trabajos que tenemos que realizar con una prioridad específica. Puede ser por ejemplo una cola de impresión donde se ordenan los trabajos en función de alguna prioridad. Para esto ordenamos los trabajos según la prioridad y luego se los procesa en este orden. Por supuesto que hay que tomar recaudos al agregar nuevos elementos y mantener la lista ordenada.
- Una técnica muy utilizada para agrupar ítems es la denominada cortes de control. Por ejemplo, si se desea obtener los totales de sueldo por departamento en una organización, ordenamos los datos por el departamento que, se convierte en el elemento de control y luego cada vez que cambia imprimimos un total. Esto se denomina un corte de control. Pueden existir varios cortes de control, por ejemplo sección, departamento, etc.
- Si deseamos hallar la mediana de un conjunto de datos podemos indicar que la mediana está en el medio. Por ejemplo para hallar el elemento k ésimo mayor bastaría en acceder a $x[k]$.
- Cuando queremos contar frecuencias y desconocemos el número de elementos existentes o este es muy grande, ordenando y aplicando el concepto de control, con un solo barrido podemos listar todos las frecuencias.

- Para realizar unión de conjuntos se pueden ordenar los conjuntos, realizar un proceso de intercalación. Cuando hay dos elementos iguales solo insertamos uno eliminando los duplicados.
- En el caso de la intersección intercalamos los dos conjuntos ordenados y dejamos uno solo de los que existan en ambos conjuntos.
- Para reconstruir el orden original es necesario crear un campo adicional que mantenga el original para realizar una nueva ordenación para, reconstruir el orden original.
- Como vimos también se pueden ordenar los datos para realizar búsquedas rápidas.

11.4. Clasificación en Java

Los métodos propios del Java proveen rutinas para ordenar objetos que facilitan el desarrollo de aplicaciones.

Para ordenar vectores que pueden ser de tipos int, long, short, char, byte, float o double es suficiente utilizar la clase *Arrays* con el método *sort*, el siguiente ejemplo genera 10 números al azar y los ordena.

```
import java.util.*;
/**
* Programa ordenar enteros
* utilizando el metodo de java
* @author Jorge Teran
*/
public class SortVect {
    public static void main(String[] args) {
        int[] x = new int[10];
        Random gen = new Random();
       //Generar 10 números enteros entre 0 y 100
        for (int i = 0; i <10; i++)
            x[i] = gen.nextInt(100);
       //Ordenar en forma ascendente
        Arrays.sort(x);
       //mostrar los números ordenados
        for (int i = 0; i <10; i++)
            System.out.println(x[i]);
        }
}
```

Cuando se requiere ordenar un objeto se hace necesario crear un mecanismo para comparar los elementos del objeto. Construyamos una clase *Persona* que almacene nombre, apellido y nota.

```
import java.util.*;

class Persona {
   private String nombre;
   private String apellido;
   private int nota;
   public Persona(String no, String ap, int n) {
      nombre = no;
      apellido = ap;
```

```
        nota = n;
    }

    public String getNombre() {
        return nombre;
    }
    public String getApellido() {
        return apellido;
    }

    public int getNota() {
        return nota;
    }
}
```

Para crear diferentes instancias de *Persona* se procede como sigue:

```
Persona[] alumnos = new Persona[3];
alumnos[0] = new Persona("Jose","Meriles", 70);
alumnos[1] = new Persona("Maria","Choque",55);
alumnos[2] = new Persona("Laura","Laura", 85);
```

Si se requiere ordenar éstos alumnos por nota no es posible utilizar directamente *Arrays.sort(alumnos)* dado que el método de clasificación no conoce cuáles y qué campos comparar para realizar el ordenamiento.

Por esto es necesario primeramente escribir un método que resuelva el problema. En nuestro ejemplo creamos el método *compareTo*. Este nombre es un nombre reservado de Java y lo que realiza es una sobrecarga de método reemplazando el método de Java con el nuestro.

Los valores que debe devolver son -1 cuando es menor, 0 cuando es igual y 1 cuando es mayor. Lo que hace este método es comparar el valor presente con otro pasado a la rutina. Esta rutina utiliza el método qsort para ordenar los valores. Esta rutina queda como sigue:

```
public int compareTo(Persona otro){
    if (nota < otro.nota) return -1;
    if (nota > otro.nota) return 1;
    return 0;
 }
```

Además es necesario especificar que la clase persona incluye el método *Comparable* y se especifica así:

```
class Persona implements Comparable<Persona>
```

El programa final queda implementado en el siguiente código:

```
import java.util.*;
/**
* Programa ordenar objetos
* utilizando el metodo de java
* @author Jorge Teran
*/
```

```
public class OrdObjeto {
     public static void main(String[] args) {
         Persona[] alumnos = new Persona[3];

         alumnos[0] = new Persona(``Jose", ``Meriles", 70);
         alumnos[1] = new Persona(``Maria", ``Choque", 55);
         alumnos[2] = new Persona(``Laura", ``Laura", 85);
         Arrays.sort(alumnos);

         for (Persona e : alumnos)
             System.out.println(``Nombre=" + e.getNombre()
         + ``, Apellido= " + e.getApellido()
         + ``, nota=" + e.getNota());
         }
}

class Persona implements Comparable<Persona>{

     private String nombre;

     private String apellido;

     private int nota;

     public Persona(String no, String ap, int n) {
         nombre = no;
         apellido = ap;
         nota = n;
         }

     public String getNombre() {
         return nombre;
         }

     public String getApellido() {
         return apellido;
         }

     public int getNota() {
         return nota;
         }

     public int compareTo(Persona otro) {
         if (nota <otro.nota)
             return -1;
         if (nota >otro.nota)
             return 1;
         return 0;
         }
}
```

11.5. Algoritmos de clasificación

Dado que el lenguaje ya incluye un soporte para realizar clasificaciones con un algoritmo parece que no es necesario revisar algoritmos de clasificación. En la realidad existen problemas que pueden ser resueltos más eficientemente con un algoritmo específico. Esto se logra conociendo como están los datos que pretendemos procesar.

Los algoritmos para ordenar se clasifican en algoritmos generales y particulares. Los algoritmos generales se pueden aplicar sin importarnos como son los datos, en cambio los algoritmos particulares son aplicables a casos especiales para obtener un mejor tiempo de proceso.

Entre los algoritmos generales se explican los métodos de clasificación por inserción, selección, qsort, y el de burbuja.

Para entender los diferentes algoritmos de ordenar es necesario comprender como establecer la invariante en cada uno de los métodos. Para esto consideraremos que los datos a ordenar corresponden al eje y de un gráfico bidimencional y la posición del vector al eje x. Ver la figura 11.1.

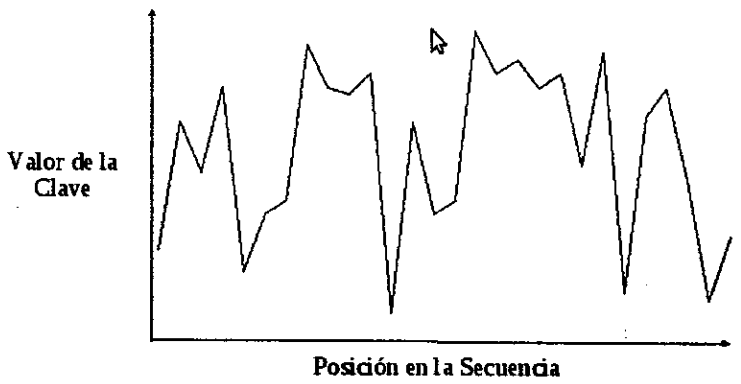


Figura 11.1: Representación de una secuencia de datos

Este gráfico representa los datos iniciales, es decir, la precondición que viene dada por la figura 11.2:

Una vez ordenados todos los elementos, la figura 11.3 que representa a la postcondición es la siguiente:

11.5.1. Método de la burbuja

El método de la burbuja es el más simple y el más antiguo. También hay que mencionar que es el método más lento para ordenar.

El método consiste en comparar todos los elementos de la lista con el elemento de su lado. Si se requiere se realiza un intercambio para que estén en el orden previsto. Este proceso se repite hasta que todos los elementos estén ordenados. Vale decir que, en alguna pasada no se realiza ningún intercambio.

El código para este algoritmo es el siguiente:

```
void Burbuja (int[] x) {
     int i, j, temp;
     int n=x.length;
     for (i = (n - 1); i >= 0; i--) {
```

Figura 11.2: Precondición de una secuencia de datos

```
for (j = 1; j <= i; j++) {
    if (x[j - 1] >x[j]) {
        temp = x[j-1];
        x[j-1] = x[j];
        x[j] = temp; }
    }
}
```

Como se ve en el programa en el primer ciclo se recorren todos los elementos. El segundo ciclo se encarga de que el último elemento sea el mayor de todos.

Analizando el programa se puede determinar que, el tiempo de proceso en el caso general es proporcional al cuadrado de la cantidad de elementos. Podemos mencionar como una ventaja la simplicidad de la codificación y como desventaja la ineficiencia del algoritmo.

Existen otros métodos que también tienen un tiempo similar pero que son mucho más eficientes que mencionamos a continuación.

11.5.2. Clasificación por inserción

Cuando se tiene un conjunto de cartas en la mano lo que hacemos es tomar una carta y buscar su ubicación e insertarla en su sitio hasta ordenar todas. Para ésto podemos suponer que el primer elemento está en su lugar y proceder a colocar los siguientes en función del primer elemento.

Representando gráficamente el método obtenemos (ver figura 11.4): La invariante indica que los valores hasta i están ordenados y los datos posteriores pueden ser menores o mayores que el último dato ya ordenado.

```
void Insercion(int[] x) {
     int i, j, temp;
     int n = x.length;
     for (i = 1; i <n; i++) {
         //los datos estan ordenados hasta i
         for (j = i; j >0 && x[j - 1] >x[j]; j--) {
```

Postcondición:

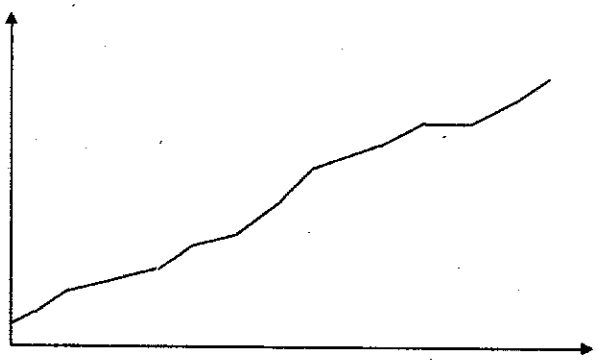


Figura 11.3: Postcondición de una secuencia de datos

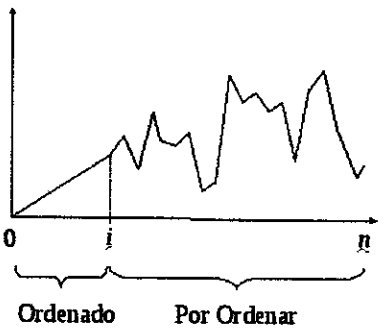


Figura 11.4: Invariante de clasificación por inserción

```
            temp = x[j-1];
            x[j-1] = x[j];
            x[j] = temp;
            }
        }
}
```

Para explicar el funcionamiento del algoritmo supongamos que inicialmente tenemos la siguiente secuencia de números

62 46 97 0 30

Inicialmente suponemos que el primer valor es el menor y lo insertamos en la primera posición que, es el lugar donde estaba inicialmente, luego se toma el siguiente valor y se ve si le corresponde estar antes o después, en el ejemplo se produce un intercambio

46 62 97 0 30

En la siguiente iteración tomamos el 97 y se ve que, no debe recorrerse hacia adelante por lo que

no se hace ningún intercambio. Luego se realiza lo mismo con el siguiente elemento recorriendo todo el vector e insertando el elemento en el lugar apropiado, obteniendo

0 46 62 97 30

esto se repite hasta obtener el vector ordenado.

La ventaja de este algoritmo es que es más simple que, el anterior y aún cuando su tiempo de proceso es también proporcional a n^2 es más eficiente y puede afinarse para ser más eficiente. El código siguiente muestra el algoritmo mejorado para ser más eficiente. El tiempo de proceso sigue siendo proporcional a $O(n^2)$

```
\begin{center}
\begin{verbatim}
void Insercion2(int[] x) {
     int i, j, temp;
     int n = x.length;
     for (i = 1; i <n; i++) {
         //los datos estanordenados hasta i
         temp=x[i];
         for (j = i; j >0 && x[j - 1] >temp; j--) {
             x[j] = x[j-1];
             }
         x[j] = temp;
         // con esta asignacion restablecemos la invariante
         }
     }
\end{verbatim}
\end{center}
```

Como se observa se ha mejorado la parte que corresponde a construir el espacio para introducir el elemento siguiente reduciendo el número de intercambios.

11.5.3. Ordenación por selección

En el algoritmo de inserción la invariante indica que los valores posteriores al índice i pueden ser mayores o menores que el último valor ordenado. Podemos cambiar esta invariante haciendo que el último valor ordenado sea menor que todos los elementos por ordenar. Esto se representa gráficamente como sigue (ver figura 11.5):

Para implementar el concepto expresado en la invariante se escoge el elemento más pequeño y se reemplaza por el primer elemento, luego repetir el proceso para el segundo, tercero y así sucesivamente. El resultado es el siguiente programa void Seleccion(int[] x) int i, j; int min, temp; int n = x.length; for (i = 0; i ¡n - 1; i++) min = i; for (j = i + 1; j ¡n; j++) if (x[j] ¡x[min]) min = j; temp = x[i]; x[i] = x[min]; x[min] = temp;

Como se observa este algoritmo es similar al algoritmo de inserción con tiempo similar y proporcional a $O(n^2)$

11.5.4. Algoritmo de clasificación rápida

El qsort cuyo nombre en inglés es QuickSort o también denominado algoritmo de clasificación rápida ya fue publicada por C.A.R. Hoare en 1962 y se basa en dividir el vector en dos mitades utilizando un elemento denominado pivote de tal forma que todos los elementos mayores al pivote

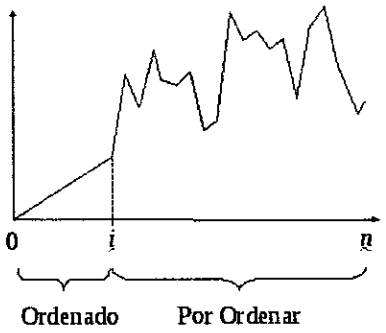


Figura 11.5: Invariante de clasificación por selección

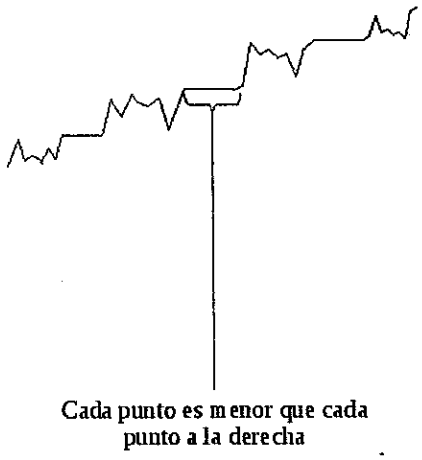


Figura 11.6: Invariante de clasificación rápida

estén en un vector y los restantes en el segundo vector. Como pivote inicial se puede tomar el primer elemento o uno al azar.

El siguiente la figura 11.6 muestra la invariante, vemos que para cada punto los valores de la derecha siempre son mayores.

Este proceso se repite en forma recursiva hasta tener el vector ordenado. A partir de este enfoque podemos realizar una primera aproximación a lo que viene ser el algoritmo:

```
Ordenar (l,u){
  if (u >= l) {
    hay como maximo un elemento
    por lo tanto no hacer nada
  }
  hallar un pivote p para dividir en vector en dos
  Ordenar (l,p-1)
  Ordenar (p+1,u)
}
```

Aquí como dijimos el problema es encontrar el punto de partición denominado pivote. Utilizando el concepto que la invariante es que: los valores entre x[m] y x[i] son menores al pivote x[l] podemos construir el siguiente código para hallar el pivote.

```
int m = l;
//invariante los valores entre x[m] y x[i]
//son menores al pivote x[l]
for (int i = l + 1; i <= u; i++) {
      if (x[i] < x[l])
      {
        temp = x[++m];
        x[m] = x[i];
        x[i] = temp;
      }
    }
temp = x[l];
    x[l] = x[m];
    x[m] = temp;
```

Supongamos que tenemos los siguientes datos:

27 2 81 81 11 87 80 7

Iniciamos un contador m en el primer valor y tomando $x[l] = 27$ como pivote y se recorre el vector comparando cada uno de los valores con el pivote haciendo un intercambio entre (m+1) y la posición del vector, obteniendo en el primer intercambio entre el número 2 que es menor que el pivote y el número de la posición m que es casualmente el 2, obteniendo:

27 2 81 81 11 87 80 7

El segundo intercambio es entre el 11 que es menor que el pivote y el 81 obteniento:

27 2 11 81 81 87 80 7

El tercer intercambio se produce con el 7 y el 81 obteniendo

27 2 11 7 81 87 80 81

Para finalizar intercambiamos el pivote con la posición del último número menor al pivote, obteniendo un vector donde todos los elementos menores al pivote están a la izquierda y los mayores a la derecha.

7 2 11 27 81 87 80 81

Este proceso se puede repetir recursivamente para el vector a la izquierda del pivote y para el que está a la derecha obteniendo finalmente:

```
  private void qsort(int[] x, int l, int u) {
    if (l >= u)
      return;
    int m = l, temp;
//invariante los valores entre x[m] y x[i]
// son menores al pivote x[l]
    for (int i = l + 1; i <= u; i++) {
```

```
      if (x[i] < x[l])
      {
        temp = x[++m];
        x[m] = x[i];
        x[i] = temp;
      }
    }
    temp = x[l];
    x[l] = x[m];
    x[m] = temp;
    qsort(x, l, m - 1);
    qsort(x, m + 1, u);
  }
```

La clasificación rápida es mucho más eficiente con un tiempo proporcional $n\log(n)$, que no es propósito de este texto probar.

Aunque no se divide exactamente a la mitad se ha tomado este valor para mostrar como mejora la eficiencia . En la realidad se puede mejorar en función de como está el conjunto de datos y como se elige al pivote.

En el caso de que el vector esté previamente ordenado el tiempo de proceso será mayor. El proceso de elegir un pivote aleatoriamente produce un algoritmo más estable. Es posible optimizar el método de ordenación rápida pero no se tratará en el texto. Cuando uno requiere un algoritmo de estas características normalmente se recurre a las rutinas ya implementadas en las librerías del lenguaje.

Se pueden mejorar aún más los algoritmos de clasificación? En casos particulares se puede obtener algoritmos proporcionales a $O(n)$ en el caso más general, el mejor algoritmo es proporcional a $n\log(n)$. Una demostración se puede leer en el texto de [GB96]

11.5.5. Algoritmos lineales

Considere las siguientes condiciones para los datos de entrada, se dispone de memoria, suficiente, los datos son enteros positivos sin repetición en el rango de 0 a N, donde N es el número máximo existente. Para ordenar estos números se puede utilizar cualquiera de los algoritmos descritos para ordenar.

Para mejorar el proceso de ordenar consideremos la siguiente estructura

```
BitSet a = new BitSet(max + 1);
```

Si cada posición del vector representa a uno de los números del vector se utiliza un bit para almacenar cada uno de los números. En este caso es posible leer los números de entrada y encender el bit que le corresponde. Recorriendo el vector se pueden imprimir los números a los que corresponden los bits encendidos en forma ordenada.

El resultado proporciona el siguiente algoritmo.

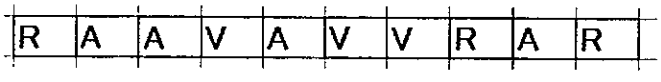
```
public void bitSort(int[] x) {
    int max=0, j=0;
    for(int i =0; i<x.length; i++){
        if (x[i] >max)
            max=x[i];
        }
    BitSet a = new BitSet(max + 1);
    // Hasta aqui todos los valores de
```

```
// a estan en cero
for(int i =0;i<x.length; i++)
    a.set(x[i]);
for(int i =0;i<= max; i++){
    if (a.get(i))
        x[j++]=i ;
    }
}
```

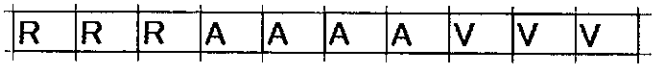
Analizando el algoritmo se ve que se tienen tres ciclos que recorren todos los datos dando un tiempo de ejecución proporcional a $O(n)$.

Como se ve en este método antes de aplicar el algoritmo ya conocemos la posición que cada elemento debe tomar. En estos casos se puede desarrollar algoritmos de tiempo lineal.

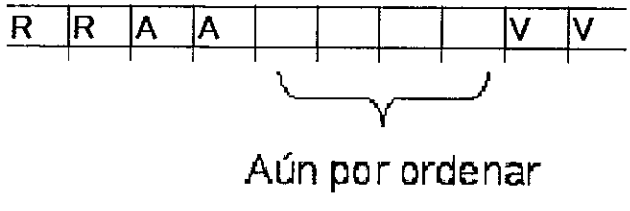
Otro ejemplo de clasificación en tiempo lineal es el problema de ordenar la bandera de Bolivia. Supongamos que tenemos una bandera que tiene los colores desordenados (precondición), tal como muestra la imagen:



Una vez ordenado (postcondición) el resultado a obtener sería



Para ordenar utilizaremos la siguiente invariante:



Para resolver este problema ya conocemos el lugar exacto de cada uno de los colores de la bandera. Rojos a la izquierda, amarillos al centro, y verdes a la derecha. El proceso que se sigue, manteniendo la invariante que nos dice que desde el último amarillo hasta el primer verde no está ordenado.

Llevamos sucesivamente los rojos a la izquierda y los verdes a la derecha. Lo amarillos quedan al centro por lo que queda ordenado. El programa java ejemplifica el proceso:

```
/**
* Programa para ordenar la bandera de Bolivia
*
* @author Jorge Teran
*
*/
public class Bandera {
     public static final void main(String[] args)
     throws Exception {
```

```
        char[] bandera = { 'R', Á', Á', 'V',
            Á', 'V', 'V', 'R', Á', 'R'};
        for (int i = 0; i <10; i++)
            System.out.print(bandera[i] + '' ");
        System.out.println('' ");
        int r = 0, a = 0, v = 10;
        char t;
        while (a != v) {
            if (bandera[a] == 'R') {
                t = bandera[r];
                bandera[r] = bandera[a];
                bandera[a] = t;
                r++;
                a++;
                } else if (bandera[a] == Á') {
                a++;
                } else if (bandera[a] == 'V') {
                t = bandera[a];
                bandera[a] = bandera[v - 1];
                bandera[v - 1] = t;
                v--;
                }
            }

        for (int i = 0; i <10; i++)
            System.out.print(bandera[i] + '' ");
        System.out.println('' ");
        }
}
```

Está probado que cualquier algoritmo basado en comparaciones toma un tiempo $O(n \log n)$. ¿Cómo es posible que ordenar la bandera tome un tiempo lineal? La respuesta es que en el proceso conocemos la posición de los elementos de la bandera.

11.5.6. Laboratorio

Con la finalidad de probar experimentalmente los algoritmos de clasificación implemente los algoritmos y llene el cuadro siguiente:

Tamaño	10	10^2	10^3	10^4	10^5	10^6
Burbuja						
Inserción						
Inserción2						
Selección						
Quicksort						
Del Java						
Bitsort						

11.6. Ejemplo de aplicación

La unidad de contabilidad tiene una cantidad de comprobantes c, que están numerados desde $0 \leq c \leq 1000$. Le han pedido leer todos los números de comprobantes $0 \leq n \leq 10^10$ y contar cuantos números de comprobantes están duplicados. La entrada de datos consiste de varios casos de prueba. Cada caso de prueba comienza con un número n entero que indica la cantidad de comprobantes. Luego vienen n números de comprobantes. Los casos de prueba terminan cuando n es cero.

Ejemplo de entrada

```
10
12 88 77 3 50 12 86 77 3 55
5
1 2 3 4 5
6
1 1 1 1 1 1
0
```

Ejemplo de salida

```
3
0
5
```

Para resolver el problema primero debemos leer en un vector los números de comprobantes. Seguidamente ordenamos el vector. Como ahora están ordenados es suficiente comparar un valor con el anterior. Si son iguales están duplicados. Después de recorrer el vector se imprime la cuenta. El programa siguiente resuelve el problema.

```
import java.util.*;
public class programa2 {

    public static void main(String[] args) {
        Scanner lee = new Scanner(System.in);
        for (int i=lee.nextInt();i>0;i=lee.nextInt()){
            int [] n = new int[i];
            for(int j=0;j<i;j++)
                n[j]=lee.nextInt();
            Arrays.sort(n); //ordenar
            //comparar un elemento con el anterior
            int contar=0;
            for (int j=1;j<i;j++){
                if (n[j]==n[j-1])
                    contar++;
            }
            System.out.println(contar);
        }
    }
}
```

11.7. Ejercicios

1. Avion o Carretera
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   José se fué de viaje por trabajo por un tiempo muy prolongado. Quiere retornar a la casa de su amada los más rápido posible. El se encuentra en la ciudad 0 y su amada en la ciudad N

   Existen carreteras y vuelos que conectan la $ciudad[i]$ a la $ciudad[i+1]$ para cualquier $0 \leq i \leq N-1$.

   Se tienen los siguientes datos tiempo de viaje por tierra que representa el tiempo que toma ir de ciudad i a la ciudad $i+1$. Tiempo de vuelo de la ciudad i a las ciudad $i+1$. Debido a limitaciones económicas José solo puede hacer K vuelos. El resto debe hacerse por tierra.

   Se quiere hallar el tiempo mínimo que demorará en encontrar a su amada.

   Por ejemplo si tenemos 3 ciudades con tiempo de viaje terrestre $4, 6, 7$, significa que para ir de la ciudad 0 a la ciudad 1 se demora 4, similarmente para ir de la ciudad 1 a las ciudad 2 se demora 6 y así sucesivamente. Ahora supongamos que el tiempo de viaje por avión es $1, 2, 3$

   Si solo puede realizar un viaje de avión, solo tiene dos rutas óptimas $0 \Longrightarrow 1$ y $1 \Longrightarrow 2$ por vía terrestre y $2 \Longrightarrow 3$ por avión. Esto toma un tiempo $4+6+3 = 13$. Si tomamos $0 \Longrightarrow 1$ terrestre $1 \Longrightarrow 2$ por avión $2 \Longrightarrow 3$ terrestre. Esto toma un tiempo $4+2+7 = 13$.

Input

Los datos de entrada comprenden múltiples casos de prueba. Cada caso de prueba comprende $1 \leq N \leq 50$ ciudades. La primera línea contiene el número de ciudades. La segunda línea tiene los tiempos para ir de la ciudad i a la ciudad $i+1$ vía terrestre todos los elementos separados por un espacio. La tercera línea contiene los tiempos para ir a las ciudades por vía aérea cada elemento separado por un espacio. La cuarta línea contiene un solo entero $1 \leq K \leq N$ que indica cuantos recorridos puede hacer en avión. La entrada termina cuando no hay más datos de prueba.

Output

Por cada caso de prueba imprima una línea con el tiempo mínimo que requerirá para llegar a la última ciudad.

Ejemplo de entrada	Ejemplo de salida
3 4 6 7 1 2 3 1 3 4 6 7 1 2 3 2 3 1 2 3 2 3 4 2 7 50 287 621 266 224 68 636 797 661 644 102 114 452 420 2	13 9 6 1772

2. Seguridad Obsesiva
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Juan esta obsesionado con la seguridad, El está escribiendo una carta a su amigo Brus y quiere que ninguna persona sea capaz de leer la misma. El utiliza un cifrado simple por substitución. Cada letra de su mensaje es remplazada con la letra correspondiente en el alfabeto de substitución. Un alfabeto de substitución es una permutación de todas las letras del alfabeto original. En este problema el alfabeto original consiste de solo de las letras minúsculas de $a - z$.

Por ejemplo si el mensaje de Juan es *hello* y su cifrado cambia $h \rightarrow q$, $e \rightarrow w$, $l \rightarrow e$, y $o \rightarrow r$, el mensaje obtenido sera *qweer*.

Si el mensaje cambia $h \rightarrow a$, $e \rightarrow b$, $l \rightarrow c$, y $o \rightarrow d$, el mensaje obtenido sera *abccd*.

Dado el mensaje original, se le pide determinar el cifrado que producirá la cadena codificada que viene primero de acuerdo al orden alfabético. En la descripción anterior, la próxima al orden alfabético es la segunda *abccd*.

Input

La entrada consiste de varios casos de prueba, Cada caso de prueba es una cadena C que viene en una sola línea ($1 \leq C \leq 50$). La entrada termina cuando no hay mas datos.

Output

La salida es una cadena de la misma longitud de la entrada codificada como se describió, escrita en una sola línea.

Ejemplo de entrada	**Ejemplo de salida**
`hello`	`abccd`
`abcd`	`abcd`
`topcoder`	`abcdbefg`
`encryption`	`abcdefghib`

3. Ordenando Vectores
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Dados dos arreglos de números enteros A, B donde cada uno contiene $(1 \leq N \leq 100)$ números. definimos la función

$$S = \sum_{i=0}^{i=N} a_i b_i$$

Se pide reordenar el arreglo A de tal forma que la función S de el valor mínimo.

Input

La entrada consiste de varios casos de prueba. Cada caso de prueba consiste de tres líneas. La primera línea tiene el número N de elementos de los vectores A, B. La segunda línea tiene los elementos del vector A separados por un espacio. La tercera línea los elementos del vector B separados por un espacio. La entrada termina cuando no hay más datos.

Output

En la salida escriba en una línea el valor mínimo de S.

Ejemplo de entrada	**Ejemplo de salida**
3 1 1 3 10 30 20 5 1 1 1 6 0 2 7 8 3 1 9 5 15 100 31 39 0 0 3 26 11 12 13 2 3 4 5 9 1	80 18 528

4. Colección Estampillas
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Su pasa tiempo es coleccionar sellos postales (estampillas). Existen N diferentes estampillas numeradas desde 0 hasta $N-1$ donde el elemento i representa el precio del sello postal i.

Su objetivo como todo coleccionista es el de tener la mayor cantidad de sellos postales posible.

Se dan dos vectores el de las estampillas que uno tiene y el vector de valores de los sellos postales. Por ejemplo: Valor de los sellos $4, 13, 9, 1, 5$, sellos que posee $1, 3, 2$. Esto significa que posee la estampilla 1 cuyo valor es 13, la estampilla 3 con valor 1 y la estampilla 2 con valor 9. Con este dinero $13 + 1 + 9 = 23$ solo puedes comprar 4 sellos postales.

Si no tienes sellos postales no puedes comprar nada.

Input

La entrada consiste en varios casos de prueba. Cada caso de prueba tiene cuatro líneas. La primera línea contiene la cantidad de sellos postales disponibles $1 \leq N \leq 50$. La segunda línea contiene separados por un espacio los precios de los N sellos postales. La tercera línea tiene la cantidad de estampillas que usted tiene ($0 \leq M \leq N$). La cuarta y última línea del caso de prueba tiene el número de estampilla que posee.

La entrada termina cuando no hay más casos de prueba.

Output

Por cada caso de prueba escriba en una línea el número de estampillas máximo que puede poseer.

Ejemplo de entrada	**Ejemplo de salida**
4 13 10 14 20 4 3 0 2 1 5 4 13 9 1 5 3 1 3 2 4 7 5 9 7 0 10 16 32 13 2 17 10 8 8 20 17 6 7 0 4 1 6 8	4 4 0 8

5. Segmentos Solapados
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

En el altiplano ha llegado un invierno muy crudo. Ha nevado y muchos tramos del camino están intransitables por la nieve. La unidad de prevención de accidentes te ha pedido que hagas un programa para hallar distancia total (en puntos) que esta con nieve.

Lo que se tiene para hallar la distancia son unos reportes que contienen los puntos de donde comienza la nieve y los puntos donde termina. Pero como este reporte lo hizo la caminera hay muchos segmentos que se superponen.

Los datos vienen en dos vectores A, B. El vector A contiene los puntos de inicio y el vector B los puntos de finalización.

Por ejemplo sean los vectores los siguientes $45, 100, 125, 10, 15, 35, 30, 9$ y $46, 200, 175, 20, 25, 45, 40, 10$. Los segmentos que cubren la nieve son $9 - 25, 30 - 46$ y $100 - 200$ dando 132 puntos.

Input

Los datos de entrada consisten de varios datos de prueba. Cada caso de prueba comienza con una línea que contiene un entero con el número de puntos ($1 \leq N \leq 50$). La segunda línea contiene los N puntos de inicio separados por un espacio. La tercera línea contiene los N puntos de finalización separados por un espacio. La entrada termina cuando no hay más datos.

Output

Imprima en la salida un número en cada línea, indicando la cantidad cubierta con nieve.

Ejemplo de entrada

```
3
17 85 57
33 86 84
8
45 100 125 10 15 35 30 9
46 200 175 20 25 45 40 10
17
4387 711 2510 1001 4687 3400 5254 584 284 1423 3755 929 2154 5719
1326 2368 554
7890 5075 2600 6867 7860 9789 6422 5002 4180 7086 8615 9832 4169 7188
9975 8690 1423
20
4906 5601 5087 1020 4362 2657 6257 5509 5107 5315 277 6801 2136 2921 5233
5082 497 8250 3956 5720
4930 9130 9366 2322 4687 4848 8856 6302 5496 5438 829 9053 4233 4119 9781
8034 3956 9939 4908 5928
19
51 807 943 4313 8319 3644 481 220 2161 448 465 1657 6290 22 6152 647
3185 4474 2168
1182 912 1832 7754 9557 7980 4144 3194 7129 5535 1172 2043 6437 7252 9508 4745
```

```
8313 8020 4017
20
8786 7391 201 4414 5822 5872 157 1832 7487 7518 2267 1763 3984 3102 7627
4099 524 1543 1022 3060
9905 7957 3625 6475 9314 9332 4370 8068 8295 8177 7772 2668 7191 8480 9211
4802 2625 1924 9970 4180
```

Ejemplo de salida

```
44
132
9691
9510
9535
9813
```

6. Números de Serie
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Usted amante de la música es dueño de muchas guitarras. Cada una tiene un número de serie. Para ser capaz de ubicar los números de serie rápidamente ha decidido ordenar la lista como sigue:

Cada número de serie consiste de letras mayúsculas de la $A - Z$ y dígitos del $0 - 9$. Para determinar el orden, se siguen los siguientes pasos:

- Si dos números de serie tienen diferente tamaño el de menor longitud viene primero.
- Si la suma de los dígitos de una cadena es menor que la otra se pone antes.
- En otros casos compare alfabéticamente donde las letras están antes que los números.

Por ejemplo: si la la lista de números de serie es "$ABCD$", "$145C$", "A", "$A910$", "$Z321$", primero se coloca la A porque es de menor longitud, luego como todos tienen 4 caracteres, $ABCD$ tiene la suma más pequeña 0 luego esta $Z231$ que es menor finalmente $A910$ viene después de $145C$ porque A viene después que el 1, en orden alfabético y ambos suman 10.

Input

Los datos consisten de múltiples datos de prueba. En cada línea viene un caso de prueba donde los números de serie están separados por un espacio. Los números de serie pueden ser de 1 a 50 inclusive.

Output

Por cada caso de prueba escriba una línea con los números de serie ordenados de acuerdo a las especificaciones.

Ejemplo de entrada	Ejemplo de salida
ABCD 145C A A910 Z321 Z19 Z20 A00 AA1 123ABC1 122ABC2 141 BCDE EA A 24A 001 2AA A3B A30 000 AAA A00 B AA 01 A2 13 A5	A ABCD Z321 145C A910 Z20 Z19 A00 AA1 122ABC2 123ABC1 A EA 141 24A BCDE 000 A00 AAA 001 2AA A30 A3B B AA 01 A2 13 A5

7. Contando en sort de Inserción
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Una secuencia de números distintos va a ser ordenada utilizando el método de ordenación por inserción. La ordenación por inserción funciona como sigue:

```
insertion-sort(A)
   inicializar una nueva secuencia vacía R
   para cada numero N en A en el orden original hacer:
      determinar el índice donde $i$ en $R$ debe ser insertado,
                  para que $R$ permanezca ordenado
      mueva cada elemento en $R$ con un índice mayor o igual a $i$
                  al índice siguiente para hacer un espacio
      ponga R[i]=N
   El vector R esta ordenado
```

por ejemplo una ordenación por inserción del vector $20, 40, 30, 10$ producirá los siguientes estados para R.

a) El primer elemento (índice 0) es $R = 20$

b) Insertar 40 no requiere movimientos $R = 20, 40$

c) Insertar el próximo elemento requiere que el 40 se mueva un lugar $R = 20, 30, 40$

d) El 10 debe insertarse en la posicion 0 haciendo que se recorran los elementos siguientes, para obtener finalmente el vector ordenado $R = 10, 20, 30, 40$

¿ Cuantos elementos se movieron?. Para insertar el 30 movimos el 40 una vez, para insertar el 10 tuvimos que mover el 20, 30 y 40, haciendo un total de 4 movimientos.

Dado un vector de números escribir una línea con el número de movimientos necesarios para ordenar el vector.

Input

Los datos de entrada consisten de múltiples casos de prueba. Cada caso de prueba es un vector con $1 \leq n \leq 50$ números, que están en una línea, separados por un espacio. La entrada termina cuando no hay mas datos.

Output

Por cada caso de prueba escriba una línea con la cantidad de movimientos necesarios para ordenar el vector utilizando en método descrito.

Ejemplo de entrada	Ejemplo de salida
20 40 30 10 -1 1 0 -1000 0 1000	4 1 0

8. Números en secuencia
   La lectura de datos es de teclado. Los resultados se muestran por pantalla.

   Un error común es ordenar números como si fueran cadenas. Por ejemplo la secuencia ordenada "1","174","23","578","71","9" no está adecuadamente ordenada si sus elementos se interpretan como números en lugar de cadenas.

   Dada una lista de números dados como cadenas, se quiere ordenar los mismos en forma ascendente.

Input

La entrada consiste en varios casos de prueba. Cada caso de prueba esta en una línea. Las cadenas están entre comillas y separadas por una coma. Cada línea contendrá entre 2 y 50 elementos. La entrada termina cuando no hay más datos.

Output

Por cada caso de prueba en la salida imprima en una línea los números ordenados separados por un espacio.

Ejemplo de entrada

```
"1", "174", "23", "578", "71", "9"
"172", "172", "172", "23", "23"
"183", "2", "357", "38", "446", "46", "628", "734", "741", "838"
```

Ejemplo de salida

```
1 9 23 71 174 578
23 23 172 172 172
2 38 46 183 357 446 628 734 741 838
```

9. Escoger equipos
La lectura de datos es de teclado. Los resultados se muestran por pantalla.

Se tiene una lista de jugadores con los que conformaremos equipos. Se tiene de cada jugador un número que califica su capacidad de juego. El número más grande representa un capacidad mayor

Para escoger dos equipos se seleccionan dos capitanes, el primero naturalmente escoge al que juega mejor. Luego el segundo capitán escoge de los que quedan el que juega mejor y así sucesivamente.

Veamos un ejemplo: los jugadores vienen como sigue: $5, 7, 8, 4, 2$, el primer capitán escogería el 8, el segundo el 7, así hasta que no queden jugadores. Con este proceso el equipo uno tendría a los jugadores con capacidad $8 + 5 + 2 = 15$ y el segundo equipo $7 + 4 = 11$. Se quiere mostrar en pantalla la diferencia en valor absoluto de ambas sumas $15 - 11 = 4$.

Input

La entrada consiste en múltiples casos de prueba. Cada caso de prueba contiene entre 1 y 50 números separados por un espacio en una línea. La entrada termina cuando no hay mas datos

Output

Por cada caso de prueba escriba en la salida una línea con la diferencia en valor absoluto de la suma de la capacidad de juego.

Ejemplo de entrada	**Ejemplo de salida**
5 7 8 4 2	4
100	100
1000 1000	0
9 8 7 6	2
1 5 10 1 5 10	0

Bibliografía

[Bre88] David M. Bressoud. *Factorization and Primality Testing.* Springer - Verlag, 1988.

[CSH05] Gary Cornell Cay S. Horstman. *Core Java 2 J2SE 5.0 Volume 1.* Sun Microsystems Press, 2005.

[Fis10] Robert Fisher. *Fibonacci Applications and Strategies for Traders.* Wiley, 2010.

[GB96] Paul Bratley Gilles Brassard. *Fundamentals of Algoritms.* Printice Hall, 1996.

[Gib93] Peter Giblin. *Primes and Programming.* Cambridge University Press, 1993.

[Kno10] Ron Knott. *Fibonacci Numbers and the Golden Section.* 2010.

[Pom93] Jorge Teran Pommier. *Fundamentos de Programación.* Johon Wiley, 1993.

[THCR90] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction to Algorithms.* Massachusetts Institute of Technology, 1990.

[Wel05] Davide Wells. *Prime Numbers - The Most Mysterious Figures in Math.* John Wiley, 2005.

Índice de figuras

Índice alfabético

Sobre los problemas

Los problemas de los ejercicios tienen las siguientes fuentes:

- Capítulo 1. Todos del autor.
- Capítulo 2. Todos del autor.
- Capítulo 3. Todos del autor.
- Capítulo 4. Algunos del autor y 2,3,5,6,8,10,11,12,13,16 recopilados del sitio www.topcoder.com y el 7 del nacional de programación del 2010 Bolivia.
- Capítulo 5. Algunos del autor y 2,3,4,5,6,9,10,11,12,13,15,16,17,18,21 recopilados del sitio www.topcoder.com y el 19 del nacional de programación del 2010 Bolivia.
- Capítulo 6. Todos recopilados del sitio www.topcoder.com.
- Capítulo 7. Algunos del autor, 1,2,5,7,8,11,12,13,14 recopilados del sitio www.topcoder.com y 3,4 del sitio acm.uva.es
- Capítulo 8. Todos del autor.
- Capítulo 9. Algunos del autor, 1,2,3,5,8,11,15,17,18 y 4,19,20,21 del sitio acm.uva.es
- Capítulo 10. Algunos del autor, y el 4 recopilados del sitio acm.uva.es
- Capítulo 11. Algunos del autor y 1,2,4,5,6,7,8,9 recopilados del sitio www.topcoder.com.