Olimpiada Científica Plurinacional Boliviana de Informática

Desarrollo de Problemas de la Competencia 2015

Jorge Humberto Terán Pomier

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

Registro de depósito legal: 4-4-3140-15

1º Edición -Octubre 2015
La Paz - Bolivia

Prefacio

Motivación

La Olimpiada Boliviana de Informática es una fuente para la motivación de los estudiantes de colegio para la solución de problemas con el uso de la tecnología. Los retos que se proponen desafían su creatividad y promueven el aprendizaje de algoritmos desde los más simples hasta muy avanzados proporcionando oportunidades de estudio en universidades e inserción en el mercado laboral. Esta competencia utiliza el lenguaje de programación c/c++, sin embargo en se prevee incorporar, muy probablemente, el lenguaje Java, en futuras competencias.

Los problemas presentados son retos presentados en la Olimpiada departamental, donde participan estudiantes de todo Bolivia buscando un obtener un espacio en la competencia Nacional.

Este texto no trata de presentar solución en el lenguaje c/c++ más bien presenta las soluciones un pseudo código, (español estructurado como lo llamamos algunos). Parte de los conceptos teóricos fundamentales que permiten comprender el problema y su solución. Lo que espero es que estos fundamentos ayuden a tener las bases para resolver problemas más complicados.

Contenido

los problemas que se presentan fueron propuestos por diferentes autores y se mencionan a continuación:

Problema	Autor
1. Centro Medio	Jorge Teran
2. Si es par cual es?	Leticia Blanco
3. Agregar	Jorge Teran
4. Bloqueos en la Ciudad	Vladimir Costas
5. Anagrama Primo	Jorge Teran
6. Primos en secuencia	Jorge Teran
7. Manuel y las canicas	Alex Pizarro
8. Representación Zeckendorf	Jorge Teran

Agradecimientos

Quiero agradecer a todas las personas que hacen posible que las olimpiadas de Informática sean exitosas año tras año. No mencionares ninguna persona en especifico por que son muchas. Sin embargo, va un sinsero agradecimiento al Vice ministerio de Ciencia y Tecnología, así como a la Asociación Cientifica Multidiciplinaria por el constante apoyo al desarrollo de la jubentud boliviana..

Retos presentados en la Olimpiada

Los problemas descritos pueden resolverse y enviarse al juez virtual de la Carrera de Informática de la Universidad Mayor de San Andrés, al que puede acceder desde la dirección http://jv.umsa.bo

M.Sc. Jorge Terán Pomier.

Índice general

1

Centro Medio

Dada una sucesión de números naturales comenzando en uno y terminando en X. Definimos el centro de la misma al número que cumple con la condición que los números de la izquierda suman igual a los números de la derecha.

Por ejemplo la serie $1, 2, 3, 4, 5, 6, 7, 8$ tiene la propiedad que los números a la izquierda del 6 suman $15 = 1 + 2 + 3 + 4 + 5$ y es igual a la suma de los números a su derecha $7 + 8$. Entonces decimos que el 6 es el centro medio.

Entrada

La entrada consiste de un número ($1 \leq n \leq 10^{17}$) que representa el número del último elemento de la serie. La entrada termina cuando no hay más datos.

Salida

La salida consiste de la palabra "NO"si no existe un centro medio o el número que representa el centro, como se definió.

2

Ejemplos de entrada

```
8
9
```

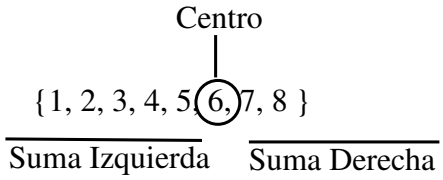
Respuestas para el ejemplo

```
6
NO
```

Centro Medio

Definición

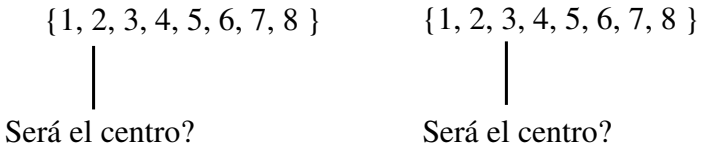
Se define el centro de una secuencia de números naturales de 1 a n, el elemento tal que, la suma de los números de la izquierda es igual a la suma de los números de la derecha. En el enunciado se indica que la conjunto está formado por la secuencia de números naturales desde 1 hasta n. La figura muestra el conjunto formado por los números del 1 al 8, vale decir $n = 8$.



En el ejemplo decimos que el centro numérico es 6 porque la suma de los números a su izquierda es igual a la suma de los números a la derecha, $1 + 2 + 3 + 4 + 5 = 15 = 7 + 8$.

Primera Idea

La primera idea que tenemos es la de ir probando cada uno de los elementos de la secuencia para ver si es el centro. Primero probamos con el primer elemento luego con el segundo y así sucesivamente.



Esto hace que tengamos que recorrer todo el vector y por cada uno de los elementos sumar todos los elementos de la izquierda y de la derecha. Si igualan es un centro. El algoritmo 1 muestra como implementar esta solución.

Esto significa que toma un tiempo de ejecución cuadrático que significa proporcional a $O(n^2)$. La siguiente ecuación muestra el número de operaciones

Algoritmo 1: Hallar el Centro

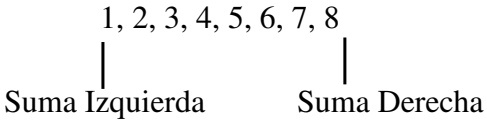
```
1 for (i=2,n)
2 |  sumaIzquierda = suma los elementos desde 1 hasta i-1 ;
3 |  sumaDerecha = suma los elementos desde i+1 hasta n ;
4 |  if (sumaIzquierda igual sumaDerecha)
5 |  |  m es el centro;
```

de suma que hay que realizar para decir si un número es centro con esta solución:

$$\sum_{i=1}^{n}\sum_{i=1}^{n} 1 = n^2$$

Una solución Lineal

Una mejor alternativa es la de tomar la suma de la izquierda como el primer elemento y la de la derecha como el último elemento:



Luego iterativamente verificamos,si la suma izquierda es menor a la suma derecha incrementamos la suma izquierda en un elemento, en el otro caso incrementamos la suma derecha. Cuando la suma izquierda y derecha igualan tenemos un centro. En este caso se ve claramente que solo se suma todos los elementos una sola vez, dando un tiempo de proceso proporcional a $O(n)$.

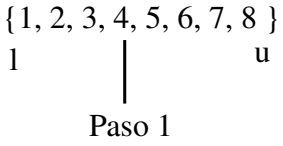
El algoritmo 2 muestra como se implementa la solución.

Mejorando la Solución

Para mejorar el tiempo de proceso nos damos cuenta que es probable que el centro este más cerca de los números más grandes. Si el primer valor es l y el último valor es u, podemos comenzar con el valor del medio:

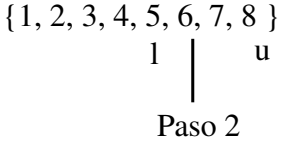
Algoritmo 2: Solución en tiempo constante

```
1  sumaIzquierda=1;
2  sumaDerecha=n;
3  l=1;
4  u=n;
5  while (sumaIzquierda < sumaDerecha)
6      if (sumaIzquierda < sumaDerecha)
7          l++;
8          sumaIzquierda+=l;
9      else
10         u--;
11         sumaDerecha+=u;
12     if (sumaIzquierda igual sumaDerecha)
13         el centro esta en l ;
```



Paso 1

Luego sumamos los valores de la izquierda y los de la derecha. Después vemos si el centro estará a la derecha o a la izquierda del punto medio. Hallamos un segundo punto medio en el lado que corresponda, en nuestro ejemplo es:



Paso 2

Sumamos los elementos tanto de la izquierda como los de la derecha y se repite hasta hallar o descartar la existencia de un punto medio. El pseudo código se muestra en el algoritmo 3.

Este proceso se denomina divide y vencerás. En cada iteración se ve que hay que sumar los n elementos. La cantidad de iteraciones ya no es igual al número de elementos, sino que se reduce a la mitad en cada una de las iteraciones.

Algoritmo 3: Hallar el centro numérico utilizando la metodología divide y vencerás

```
1  void BuscarCentro(l,u)
2      m = (l + u)/2;
3      if (l > u)
4          no tiene centro;
5      sumaIzquierda = suma los elementos desde l hasta m-1 ;
6      sumaDerecha = suma los elementos desde m+1 hasta l ;
7      if (sumaIzquierda igual sumaDerecha)
8          m es el centro;
9      if (sumaIzquierda ≤ sumaDerecha)
10         l=m-1;
11         BuscarCentro(l,u) ;
12     else
13         u=m+1;
14         BuscarCentro(l,u) ;
```

¿Cuántas iteraciones hay que realizar? Despejando i de la ecuación:

$$\frac{n}{2^i} = 1$$

Obtenemos $i = \log n$, haciendo un tiempo total proporcional a $O(n \log n)$.

Si utilizamos la formula para hallar el resultados de la suma aritmética, podemos eliminar, el proceso de hallar la suma de la izquierda y de la derecha, reducimos el tiempo a $\log n$. Recordamos que el resultado de la suma aritmética se halla con la formula:

$$\sum_{i=inicio}^{fin} i = \frac{Nterminos(inicio + fin)}{2}$$

Otra solución eficiente

Si recordamos la fórmula de la suma aritmética para los elementos del primero al último tenemos:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Ahora supongamos que el centro está en la posición x, las ecuaciones para la suma de los elementos a ambos lados es:

El lado izquierdo:

$$\sum_{i=1}^{x-1} i = \frac{(x-1)((x-1)+1)}{2} = \frac{(x-1)(x)}{2}$$

El lado derecho:

$$\sum_{i=x+1}^{x} i = \frac{(n-x)((x+1)+n)}{2} = \frac{(n-x)(n+x+1)}{2}$$

Igualando y simplificando se tiene:

$$2x^2 = n(n+1)$$

Esto significa que verificando que la raíz cuadrada de $n(n+1)/2$ es un número entero sabremos si la secuencia tiene un centro numérico y cual es. El algoritmo 4 muestra esta solución.

Algoritmo 4: Otra solución eficiente

```
1 a = u*(u+1)/2;
2 b = ⌊√a⌋;
3 if (a igual b^2)
4 |   tiene centro;
5 else
6 |   no tiene centro;
```

La interrogante que queda es saber si esta solución es mejor que la solución usando divide y vencerás que tiene un tiempo proporcional a $\log n$. Aún cuando parece una mejor solución la eficiencia depende del calculo de la raíz cuadrada. El calculo de la raíz cuadrada toma un tiempo proporcional a la precisión que se busca y el tiempo puede aproximarse a $O(\sqrt{n}$ .Por esta razón es deseable conocer si $\log n < \sqrt{n}$. Para esto verificaremos:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$$

para resolver esta expresión utilizamos la regla de L'Hopital:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/(n)}{n^{-1/2}/2} = 2 \lim_{n\to\infty} \frac{\sqrt{n}}{n} = 2 \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

Con lo que probamos que la raíz crece más rápidamente que el logaritmo, demostrando así que esta solución es eficiente, sin embargo la solución que utiliza divide y vencerás con tiempo proporcional a $\log n$ es mejor.

Extendiendo el problema

Ahora deseamos extender éste problema de la siguiente forma: dados dos números A,B donde $1 \leq A, B \leq 10^{18}$ y $A < B$ cuantos conjuntos tienen un centro medio. Para comenzar no es posible hallar todos los centros numéricos hasta 10^{18} en un tiempo razonable. Si consideramos que se pueden hacer aproximadamente 10^8 operaciones en un segundo, esto toma mucho tiempo.

Para mejorar el tiempo construyamos una tabla que tenga los primeros centros numéricos utilizando alguno de los algoritmos anteriores:

	n	**Centro**	$Centro_{i+1}/Centro_i$
0	1	1	
1	8	6	6
2	49	35	5,8333333333
3	288	204	5,8285714286
4	1681	1189	5,8284272498
5	9800	6930	5,8284271284
6	57121	40391	5,8284271284
7	332928	235416	5,8284271249
8	1940449	1372105	5,8284271247

Al dividir los centros obtenidos por el anterior vemos que se va aproximando a una constante, para resolver el problema con precisión es necesario utilizar una constante de 64 bits, es decir de doble precisión. Con estos datos ya es posible conocer cuales valores de n tienen un centro numérico. Si extendemos este concepto a los valores de n en la tabla siguiente, podemos ver que también se aproxima a una constante.

	n	$n_{i+1} - n_i$	n_{i+1}/n_i
0	1		
1	8	7	6
2	49	41	5,8571428571
3	288	239	5,8292682927
4	1681	1393	5,8284518828
5	9800	8119	5,8284278536
6	57121	47231	5,8284271462
7	332928	275807	5,8284271254
8	1940449	1607521	5,8284271248

Como vemos los centros conforman una ecuación exponencial. Una vez que tenemos esta constante calculada podemos muy fácilmente decir si un número n tiene un centro o no. En la tabla presentada podemos observar que $\lfloor 5,8284271248^7 = 1940449 \rfloor$. Ahora como se ve el es muy fácil resolver el problema.

2

¿Si es, par cuál es?

La serie de Fibonacci es una de las más conocidas en el ámbito de las matemáticas. Los primeros elementos de la serie son:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, ..$$

En esta serie se pueden hacer varias consultas como por ejemplo cuál y cómo es el $n - simo$ número Fibonacci. Si se hace la consulta cuál y cómo es el cuarto Fibonacci deberia decir "2 y es par", en cambio si se pregunta por el octavo deberá decir "13 y es impar". En realidad en este problema lo único que se quiere saber es cómo es y cuál es el dígito que lo hace par o impar.

Entonces nuevamente hacemos la consulta y le preguntamos cuál y cómo es el 4 deberá decir "2 par", si preguntamos por el 8 deberá decir "3 impar".

Entrada

La primera línea tiene un número entero c que identifica el número de casos de prueba. Cada caso de prueba está en una línea que tiene n ($1 \leq n \leq 10^{18}$) que identifica el "n-simo" Fibonacci al que se quiere preguntar cuál y cómo es.

Salida

Para cada caso de prueba, la salida debe mostrar en una línea cuál dígito hace la paridad o imparidad del $n - simo$ Fibonacci y su cualidad "par" o "impar".

Ejemplos de entrada

```
5
2
95294590
8
123
574
```

Respuestas para el ejemplo

```
1 impar
4 par
3 impar
1 impar
8 par
```

¿Si es, par cuál es?

Conceptos

Los números Fibonnaci se definen como:

$$f(x) = \begin{cases} 0, & \text{if } x = 0. \\ 1, & \text{if } x = 1. \\ f(x-1) + f(x-2), & \text{en otros casos.} \end{cases} \tag{2.1}$$

Si desarrollamos un algoritmo recursivo para hallar los números de Fibonacci como se muestra en el algoritmo 5

Algoritmo 5: Hallar los numeros Finobacci recursivamente

```
1 void Fib(x){
2     if (x igual 0)
3         return 0;
4     if (x igual 1)
5         return 1;
6     return Fib(x - 1)+Fib(x - 2) ;
```

Obtendremos un programa extremadamente lento. Para entender esto dibujamos un árbol con todos los cálculos realizados para el hallar un número Fibonacci. En la figura 2.1 vemos que muchos números se calculan varias veces haciendo de estos un proceso exponencial.

Este hecho se puede demostrar a partir de la solución de la ecuación 2.2 de recurrencia que da una expresión para calcular directamente estos números.

$$f_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n \tag{2.2}$$

Ahora en la ecuación 2.3 vemos que los números de Fobonacci son proporcionales a:

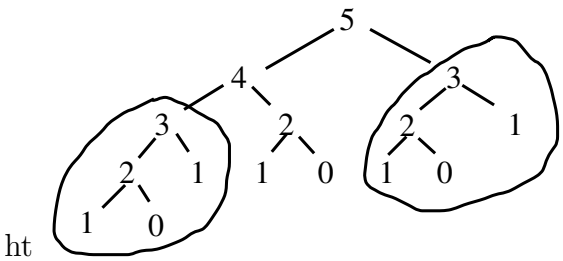


Figura 2.1: de recursión para hallar el fibonacci 5

$$f_n = (\frac{1+\sqrt{5}}{2})^n = 1{,}618^n \tag{2.3}$$

La figura 2.2 gráfica la función de función de Fibonacci mostrando su crecimiento exponencial.

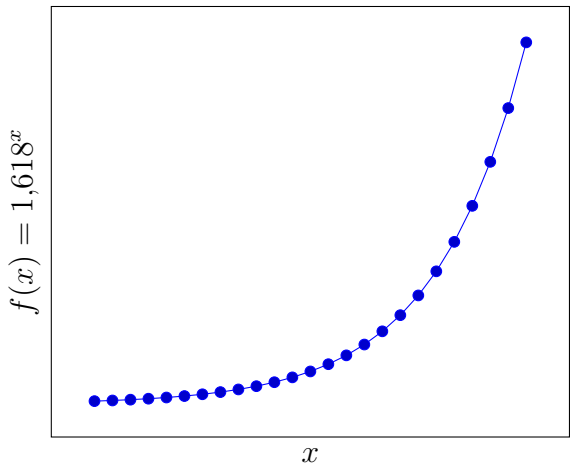


Figura 2.2: Grafica de la función de numeros Fibonacci

Hallando los números de Fibonacci

La forma más sencilla es definir un vector de enteros y en ellos calcular aplicar la formula de la recurrencia, es decir sumar los dos elementos anteriores del vector para obtener el actual, El algoritmos 6 muestra esta solución.

Algoritmo 6: Algoritmo para hallar los numeros Finobacci iterativamente

```
1 f_0 = 0;
2 f_1 = 1;
3 for (i=2,100)
4     f_i = f_{i-1} + f_{i-2}
```

Haciendo la prueba experimental del algoritmo 6 vemos que solo se puede calcular hasta el numero Fibonacci 93. Cuando tratamos de calcular el siguiente numero de la serie se produce un desbordamiento.El ultimo valor con precisión de 64 bits es 7540113804746346429

Esto hace necesario que sea necesario buscar otro método para hallar soluciones a problemas que requieren números mucho mas grandes hay que recurrir a otras tecnicas. Generalmente en los problemas se pide hallar resultados modulo un numero grande.

Aritmética Modular

La aritmética modular nos brinda la posibilidad de procesar números grandes y conservar los últimos dígitos del resultado. La operación se con el símbolo %. Las propiedades que tenemos son:

- Distributiva con respecto a la suma

$$(a+b)\,\%m = (a\,\%m + b\,\%m)\,\%m$$

- Distributiva respecto a la multiplicación

$$(a \cdot b)\,\%m = (a\,\%m \cdot b\,\%m)\,\%m$$

Para el enunciado del problema, cada vez que hallamos un número podemos hallar el módulo 10 dado que nos piden solo el último dígito

Solución secuencial

Para hallar los números de Fibonacci podemos hallar los números de Fibonaci sumando los dos números anteriores, y como se explicó tomando en cada operación el modulo 10. El algoritmo 7 muestra esta solución.

Algoritmo 7: Hallar el ultimo digito Fibonacci iterativamente

```
1 a=0;
2 b=1;
3 for (i=2,n)
4 |   c=(a+b) % 10 ;
5 |   a=b ;
6 |   b=c ;
```

Como vemos para los límites dados en el enunciado el tiempo se hará extremadamente largo.

Solución en tiempo logarítmico

Los primeros números de la secuencia de Fibonacci son:

$$\begin{bmatrix} i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ fib_i & 0 & 1 & 1 & 2 & 3 & 5 & 8 \end{bmatrix}$$

Podemos ver que el $fib_5 = 5$, el $fib_6 = 8$, etc.

Para hallar la solución en tiempo logarítmico utilizaremos la siguiente matriz

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Al multiplicar por esta matriz se obtiene el próximo número de Fibonnaci. Para multiplicar dos de estas matrices podemos utilizar el Algoritmo 8.

Ahora corresponde demostrar que la multiplicación de la matriz planteada genera la secuencia de Fibonacci, para lo cual utilizaremos el método por inducción.

Algoritmo 8: Psuedo Código para multiplicar la matriz de Fibonacci

```
1 void MulMat(int [][] a, int [][] a, int [][] res)
2     int [][] res={0,0},{0,0};
3     res[0][0]=a[0][0]*b[0][0]+a[0][1]*b[1][0];
4     res[0][1]=a[0][0]*b[0][1]+a[0][1]*b[1][1];
5     res[1][0]=a[1][0]*b[0][0]+a[1][1]*b[0][1];
6     res[1][1]=a[1][0]*b[0][1]+a[1][1]*b[1][1];
7     return res ;
```

- Probamos para los primeros números

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} fib_2 & 1 \\ 1 & 0 \end{bmatrix}$$

- Para los siguientes números de la secuencia

$$A^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} fib_3 & 1 \\ 1 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} fib_4 & fib_3 \\ 2 & 1 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} fib_5 & fib_4 \\ fin_4 & fib_1 \end{bmatrix}$$

Después de varias multiplicaciones nos damos cuenta del significado exacto de los resultados obtenidos y procedemos a la demostración en un caso general.

- Ahora consideremos un caso arbitrario j multiplicado por la matriz tenemos

$$A^{j+1} = \begin{bmatrix} fib_j & fib_{j-1} \\ fib_{j-1} & fib_{j-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} fib_j + fib_{j-1} & fib_j \\ fib_{j-1} + fib_{j-2} & fib_{j-1} \end{bmatrix} =$$

$$\begin{bmatrix} fib_{j+1} & fib_j \\ fib_j & fib_{j-1} \end{bmatrix}$$

Con lo que se demuestra que al llevar a una potencia j se obtiene el siguiente número de la secuencia.

Para reducir el tiempo lineal utilizaremos dos principios de las potencias. Si queremos hallar el valor de a^n en el caso de que n sea par podemos hacer:

$$a^n = a^{n/2}a^{n/2}$$

Si calculamos $a^{n/2}$ una sola vez y luego la elevamos al cuadrado se reducen las multiplicaciones a la mitad.

En el caso de que n sea impar n es posible hacer lo mismo y por lo tanto queda como:

$$a^n = a^{(n-1)}a$$

Aplicando esta idea se puede hallar los números Fibonacci en un tiempo proporcional a $O(\log n)$.

Solución en tiempo constante

Para mejorar esta solución es necesario conocer algunas propiedades de los números de Fibonacci. Para comenzar demostremos que la secuencia de números consiste de un número par y dos impares.

Todos los números pares pueden expresarse como $2n$ que se demuestran que todo numero par es múltiplo de 2. Por lo tanto los números impares puede expresarse como $2n + 1$. De estas propiedades podemos plantear la suma de pares e impares:

$$\text{par+par} = 2n + 2n = 4n = par$$
$$\text{par+impar} = 2n + 2n + 1 = 4n + 1 = impar$$
$$\text{impar+impar} = 2n + 1 + 2n + 1 = 4n + 2 = par$$

Consideremos por ejemplo el número 8 que se forma por la suma de $5 + 3$. El siguiente número será 8 (par) + 5 (impar) = 13 (impar). Continuando número 13 (impar) + 8 (par)= 21 (impar) y así sucesivamente.

Ahora podemos decir si un número de Fibonacci es par o impar.

$$\begin{bmatrix} i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \cdots \\ fib_i & 0 & 1 & 1 & 2 & 3 & 5 & 8 & \cdots \\ & 0 & impar & impar & par & impar & impar & par & \end{bmatrix}$$

Números de Pisano

En teoría de números, el período Pisano n, que se escribe como $\pi(n)$, es el período con el cual la secuencia de los números de Fibonacci tomada módulo n se repite. Existen diferentes períodos, dependiendo del valor n que se tome para hallar el módulo. El cuadro 2.1 muestra los resultados de aplicar el módulo n a la secuencia de Fibonacci.

n	$\pi(n)$	periodos
1	1	0
2	3	011
3	8	0112 0221
4	6	011231
5	20	01123 03314 04432 02241
6	24	011235213415 055431453251
7	16	01123516 06654261
8	12	011235 055271
9	24	011235843718 088764156281
10	60	011235831459437 077415617853819 099875279651673 033695493257291

Cuadro 2.1: Los primeros 10 números de Pisano

De donde sabemos que la propiedad de los números Fibonacci es que el último dígito se repite cada 60 números, cuando aplicamos el módulo 10.

Con esto es suficiente hallar los primeros 60 últimos dígitos de los números de Fibonacci, usando un algoritmo lineal y luego hallando el módulo 60 sabemos que dígito será el último.

Extendiendo el problema

Una extension natural del problema es calcular los períodos de Pisano. En este caso utilizando la aritmética modular podemos hallar periodos de 1000 y más dígitos. A continuación se muestra un ejemplo, tomando el modulo 4.

$$\begin{bmatrix} fib & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 \\ fib\,\%4 & 0 & 1 & 1 & 2 & 3 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Como podemos hallar el período $\pi(n)$. Para esto simplemente hallamos los restos después de hallar el módulo n. Luego verificamos donde comienza a repetirse la secuencia. El algoritmo 9 nos muestra como podemos hallar esta repetición.

Algoritmo 9: Algoritmo para hallar periodo Pisano

```
1 for (i=2,93)
2     fibMod[i]=fib[i] % n ;
3 for (i=2,93)
4     if (fibMod_i igual fibMod_0 y
5     (fibMod_{i+1} igual fibMod_1) y
6     (fibMod_{i+2} igual fibMod_2)
7         imprimir "π(n) = i";
```

3

Agregar

El nombre del problema refleja la tarea, sumar un conjunto de números. Usted puede estar pensando en un programa que solo sume los números, pero hay una pequeña condición que hay que cumplir.

La operación de adición requiere un resultado *costo actual* que es la suma de los operandos. Por ejemplo si queremos sumar los elementos del conjunto 1,2 y 3 hay varias formas de hacerlo en pares de dos:

```
Forma 1
1 + 2 = 3, costo actual = 3
3 + 3 = 6, costo actual = 6
Total = 9

Forma 2
1 + 3 = 4, costo actual = 4
2 + 4 = 6, costo actual = 6
Total = 10

Forma 3
2 + 3 = 5, costo actual = 5
1 + 5 = 6, costo actual = 6
Total = 11
```

Espero que haya entendido nuestra misión y sumar el conjunto de enteros

para que el costo sea mínimo.

Entrada

Cada caso de prueba comienza con un número entero positivo N ($2 \leq N \leq 10000$) seguido de N números enteros positivos menores a 100,000. La entrada termina cuando el valor de N es cero.

Salida

Por cada caso de entrada imprima el costo mínimo de adición en una sola línea.

Ejemplos de entrada

```
3
1 2 3
4
1 2 3 4
0
```

Respuestas para el ejemplo

```
9
19
```

Problema Agregar

En el problema nos piden sumar dos elementos de un conjunto, quitar los elementos del conjunto e insertar la suma al conjunto. Cuando queda un solo elemento tenemos la respuesta. Lo que se quiere es que este resultado de la suma más pequeña (mínima).

Solución Inicial

Para esto una primera alternativa es buscar los dos elementos más pequeños del conjunto en forma secuencial. Esto implica recorrer todo el vector los que significa un tiempo proporcional a n. Luego eliminar los elementos escogidos también toma un tiempo proporcional a n, insertar la suma conjunto en un tiempo constante dado que es posible conocer siempre el último elemento del vector. Esto proporciona un tiempo total que es lineal.

En el ejemplo del problema tenemos los números $3, 1, 2$ y escogemos los dos más pequeños como se muestra en la figura 3.1.

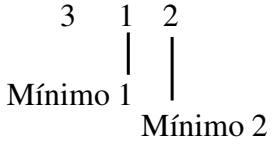


Figura 3.1: Solucion del ejemplo - primera iteración

Se extraen estos dos valores, se suman y luego se incluyen a la lista, el resultado se muestra en la figura 3.2

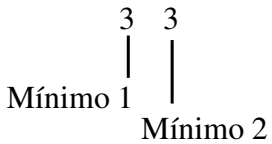


Figura 3.2: Solucion del ejemplo - segunda iteración

Finalmente se repite el proceso de buscar los dos mínimos, que luego de sumarlos nos da el resultado final.

Solución Eficiente

Ahora implementaremos una solución más eficiente, para esto consideraremos una estructura de datos conocida como *montículo*.

Un montículo es una estructura tipo árbol, que se utiliza para representar elementos, pero en los ejemplos representaremos números. Consideremos el montículo de la figura 3.3

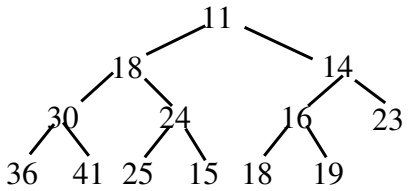


Figura 3.3: Ejemplo de montículo

Este árbol binario denominado montículo tiene dos propiedades. La primera denominada *orden*, que indica que cada nodo tiene un valor menor o igual que sus descendientes. Esto implica que el menor elemento se encuentra en la raíz. La segunda propiedad es la forma, la figura 3.4 muestra esta propiedad.

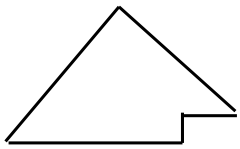


Figura 3.4: Forma de un montículo

Las características en complejidad de esta estructura de datos se ve en el cuadro 3.1.

Operación	Tiempo
Hallar el mínimo	constante
Insertar	proporcional a $\log n$
Eliminar la raíz	proporcional a $\log n$

Cuadro 3.1: Tiempo de ejecución de un montículo

La primera característica es muy simple de probar dado que por definición

el valor mínimo esta en la raíz. Para mostrar que insertar o borrar toma un tiempo proporcional a $\log n$ vemos primero el desplazamiento hacia arriba.

Supongamos que insertamos un elemento al final del montículo como se muestra en la figura 3.5

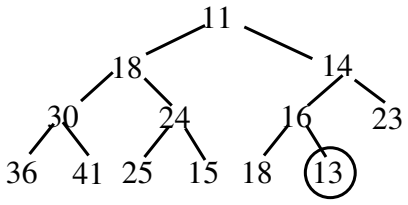


Figura 3.5: Un elemento insertado al final

Por la definición los descendientes deben ser mayores al nodo padre, por lo que es necesario intercambiar los elementos para mantener la propiedad. La figura 3.6 muestra este intercambio.

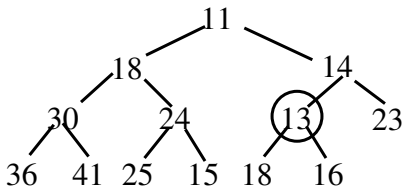


Figura 3.6: Ejemplo de intercambio de nodos

Con este intercambio vemos que el elemento sube una posición. Como aún no se cumple la condición se ve que es necesario repetir el proceso hasta satisfacer la definición, como se muestra en la figura 3.7.

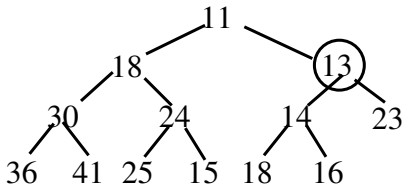


Figura 3.7: Estado final despue del intercambio

A lo sumo seguirá un camino que lo lleve a la raíz del árbol y como tiene una profundidad de $\log n$ donde es el número de nodos, este es el máximo número

de intercambios y por ende la complejidad de la operación. En la figura 3.7 se muestra el estado final.

Ahora veamos el recorrido hacia abajo, supongamos que insertamos un valor en la raíz, ver la image 3.8 . Al reorganizar el nodo padre y sus descendientes se ve como baja el elemento insertado hasta su posición final, como se muestra en las figuras 3.9 y 3.10.

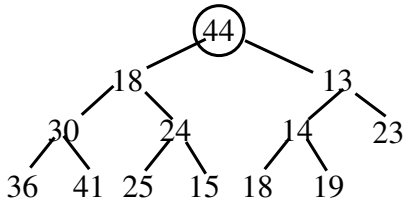


Figura 3.8: Elemento insertado en la raiz

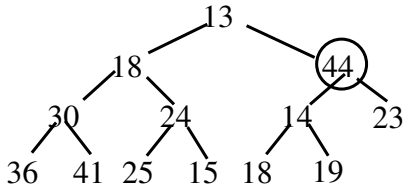


Figura 3.9: El elemento deciende despues de un intercambio

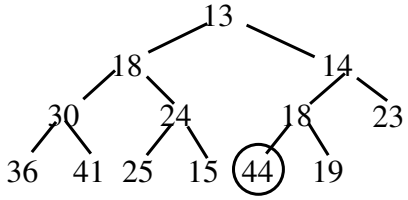


Figura 3.10: Estado final despue de descender

Cola de prioridad

Toda estructura se analiza desde dos puntos, la especificación de lo que hace y su implementación. La especificación nos dice lo que hace, la cola de prioridad implementa principalmente dos instrucciones, *extraer el mínimo* e *insertar*.

Desde el lado de la implementación trabaja con una estructura de datos de montículo. Esta es a razón por la que la cola de prioridad toma los tiempos que se muestran en el cuadro 3.1.

Operación	Tiempo
Extraer el mínimo	proporcional a $\log n$
Insertar	proporcional a $\log n$

Cuadro 3.2: Tiempo de ejecución de un montículo

El lenguaje Java y C++ proporcionan una librería que implementa la cola de prioridad.

Implementación

Para este problema la solución eficiente es definir una cola de prioridad y seguir las instrucciones de extraer e insertar como se indica en el enunciado.

Algoritmo 10: Solucion al problema agregar

```
1 Definir q como cola de prioridad;
2 Leer los datos a una cola de prioridad;
3 while (q tenga mas de un elemento)
4 |   extraer de q en a ;
5 |   extraer de q en b ;
6 |   poner a q el valor de a + b ;
```

4

Bloqueos en la Ciudad

En una ciudad bien organizada, donde las calles han sido organizadas en un cuadrado perfecto de `NxN` calles, la gente se ha vuelto loca y ha comenzado a bloquear las calles. Aparentemente no hay razón para los bloqueos.

Los bloqueos se dan en las esquinas de las calles. En la ciudad las calles se nominan con números. Las verticales se nominan desde 0 (se comienza más a la izquierda) y en horizontales desde 0 (se comienza desde arriba).

Lamentablemente el representante de la Organización de Ciudadanos Ejemplares, Prudentes y Bondadosos (OCEPB) debe trasladarse desde su casa en la esquina `(X0,Y0)` a la sede de la OCEPB en `(Xf,Yf)` donde se reunirá con un hombre que dice tener la solución al problema de la locura de la gente, este señor trabaja para la Organización de Buenos Individuos (OBI).

Ayuda al representante de la OCEPB a saber el camino para llegar a su destino conociendo el mapa actual de la ciudad y sus bloqueos. En caso de que pueda llegar debes indicar que `HAY RUTA POSIBLE`. Si fuera imposible llegar debes indicarle que `NO HAY RUTA POSIBLE`.

En caso de que la esquina de destino o la esquina de origen este bloqueada debe indicar `NO HAY RUTA POSIBLE`.

Entrada

Se tienen varios casos de prueba, cada caso de prueba consiste de varias líneas.

Primera línea `N`, Número de calles verticales y horizontales con $2 \leq N \leq 100$

Segunda línea: `X0 y0 Xf Yf`

`(X0,Y0)` es el punto origen del representante de la OCEPB, y `(Xf,Yf)` es su destino.

Siguen `N` líneas con `N` caracteres representando la posibilidad de bloqueos en la ciudad. `B` significa Bloqueado y `L` Libre

Los casos de prueba terminan cuando `N` es 0.

Salida

Una línea con dos posibles mensajes: `HAY RUTA POSIBLE`, ó `NO HAY RUTA POSIBLE`

Ejemplos de entrada

```
5
0 1 1 4
BLBLB
LLLLL
BBBLL
LBLBB
BBLLB
0
```

Respuestas para el ejemplo

```
HAY RUTA POSIBLE
```

Bloqueo

Solución por recursividad

Analizando el problema vemos que existen cuatro lugares a donde se puede mover. Si en algún momento se encuentra en la posición Pi, j puede moverse a las posiciones $Pi + 1, j$, $Pi - 1, j$, $Pi, j - 1$, $Pi, j + 1$ tal como se muestra en la figura 4.1

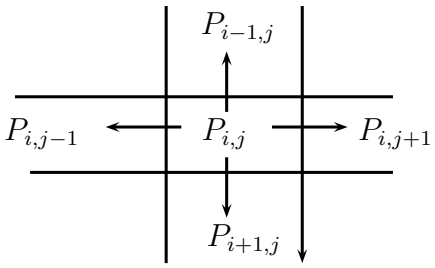


Figura 4.1: Lugares donde se puede mover de la posición Pi, j

Representando las posiciones arriba, abajo, izquierda y derecha. Con esto en mente escribimos un programa recursivo que visite estas posiciones solo cuando no estén bloqueadas. Para terminar la recursión nos fijamos si llegamos a la salida deseada.

Esta solución no garantiza que no se entre en un ciclo infinito. Podemos llegar a una dos veces a una misma posición, por ejemplo desde arriba por un camino y por abajo por otro. Para evitar que esto se convierta en un ciclo creamos una matriz para marcar loas posiciones visitadas si llegamos a una posición visitada terminamos esa recursión continuando con otra posiciones.

Un algoritmo que puede recorrer todos los elementos como se ve en el algoritmo 11.

En este algoritmo hemos creado dos arreglos uno denominado ciudad que es el que nos dan en la entrada y otro que denominamos *visitados* para anotar todos los lugares ya examinados. Esto elimina los bucles y el proceso repetido. En el algoritmo 11 no hemos controlado los extremos. Esto significa que cuando llegamos a un extremo derecho no existe la columna siguiente, por lo que se produce un error. Para evitar estar controlando estos puntos en los que nos salimos del espacio definido, es conveniente agregar un contorno

Algoritmo 11: Recorrer la ciudad recursivamente

```
1  void Buscar(x,y)
2      if ((x igual salida_x)&&(y igual salida_y))
3          return existe una salida;
4      if (visitados_{x,y} igual 1)
5          return // ya fue visitado ;
6      visitados_{x,y}=1;
7      // verificar la fila de abajo ;
8      if (ciudad_{x+1,y} igual 0)
9          void Buscar(x+1,y);
10     // verificar fila de arriba ;
11     if (ciudad_{x-1,y} igual 0)
12         void Buscar(x-1,y);
13     // verificar la columna izquierda ;
14     if (ciudad_{x,y-1} igual 0)
15         void Buscar(x,y-1);
16     // verificar la columna derecha ;
17     if (ciudad_{x,y+1} igual 0)
18         void Buscar(x,y+1);
```

que evite este problema. En el ejemplo podemos agregar un contorno de bloqueos como se muestra en la figura 4.2.

En el algoritmo 11, suponemos que se creo un contorno como el mostrado.

Solución utilizando grafos

Para entender esta solución pongamos el ejemplo que se muestra en el cuadro 4.1 en una matriz de dos por dos:

Ahora numeremos los elementos de esta matriz secuencialmente como se muestra en el cuadro 4.2:

Con esto dicho podemos representar los posibles movimientos en un grafo, los números representaran los nodos, y la distancia para ir de un nodo a otro definimos como uno, porque solo deseamos encontrar una salida. En el

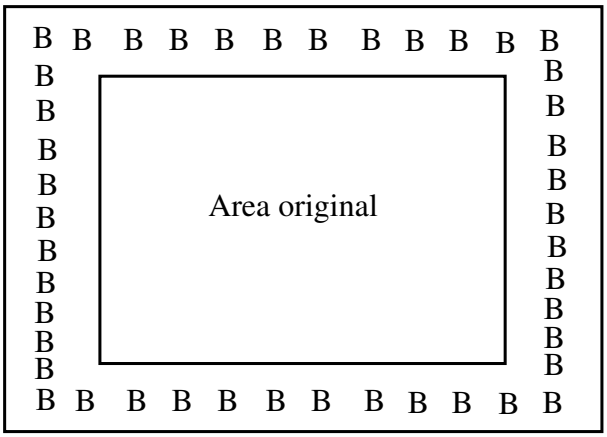


Figura 4.2: Insertar un contorno

B	L	B	L	B
L	L	L	L	L
B	B	B	L	L
L	B	L	B	B
B	B	L	L	B

Cuadro 4.1: Ejemplo de bloqueo

ejemplo se comienza en la fila 0, columna 1. En nuestra nueva nomenclatura será el nodo 1 y como se ve solo puede conectarse al nodo 6. La figura 4.3 muestra este grafo.

Para representar un grafo en una computadora utilizamos un vector donde cada elemento es una lista enlazada. Las listas enlazadas se encuentran en las librerías de todos los lenguajes de programación. El ejemplo puede representarse en un vector de las listas enlazadas. En la figura 4.4 se muestra un

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Cuadro 4.2: Numeración de las posiciones del ejemplo

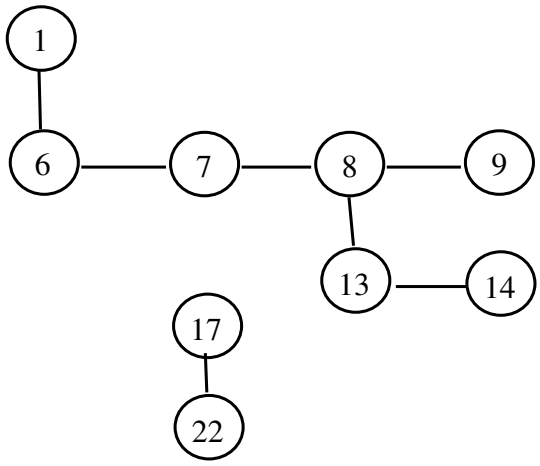


Figura 4.3: Grafo que representa el ejemplo

vector donde cada nodo es un elemento del vector y cada uno de esto apunta a los lugares donde tiene conexión.

Para el recorrido de un grafo existen dos algoritmos *Breadth-first search* (BFS) y *Depth-first search* (DFS). Estos argoritmos para el recorrido de un grafo también se denominan recorrido en anchura y en profundidad.

El recorrido en anchura recorre todos los nodos del grafo, como se muestra en el algoritmo 12. Para resolver el problema solo es necesario verificar si en el recorrido llegamos al punto de salida.

Algoritmo 12: Algoritmo Breadth-First-Search - BFS

```
1 BFS(x,y)
2     Crear una cola vacía Q ;
3     insertar el nodo inicial en Q ;
4     while (Q no está vacía)
5         sacar en U un elemento de la cola. ;
6         ∀ ( nodo n adyacente a U)
7             if (n no fue visitado)
8                 agregar n a la cola Q ;
```

Para este problema también podemos utilizar el algoritmos DFS, la diferencia

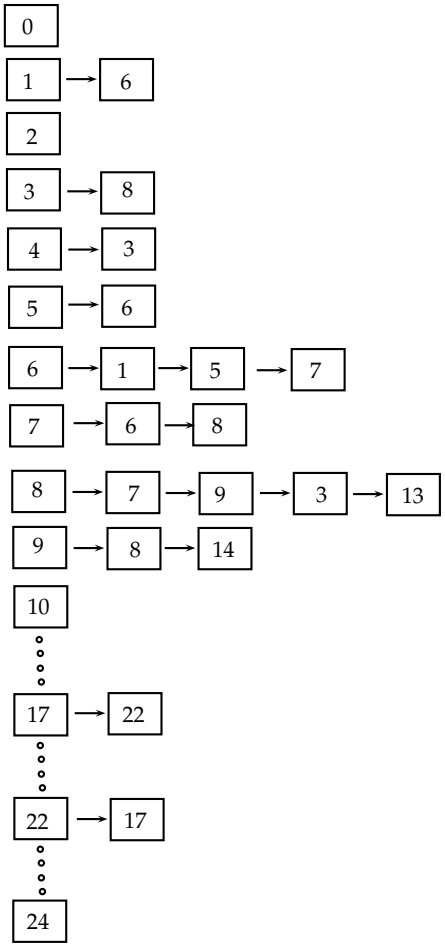


Figura 4.4: Listas enlazadas que representan el grafo

con el la solución anterior es que en lugar de utilizar una cola utilizamos una pila. El algoritmo 13 muestra esta implementación.

Algoritmo 13: Algoritmo Depth-first search - DFS

```
1 DFS(x,y)
2     Crear una pila vacía P ;
3     insertar el nodo inicial en P ;
4     while (P no está vacía)
5         sacar en U un elemento de la pila. ;
6         ∀ ( nodo n adyacente a U)
7             if (n no fue visitado)
8                 agregar n a la pila P ;
```

El la complejidad en esta representación es O(nodos+vértices). Esto se verifica porque hay que recorrer todos los nodos y en cada nodo todos sus enlaces.

Variantes al problema

Un problema adicional que se puede plantear es: ¿Cuál es el camino más corto para llegar al destino? Para resolver esto hay que revisar el algoritmos de Dijsktra.

Adicionalmente es posible pensar en cuál será el segundo camino más corto.

5

Anagrama Primo

Como sabemos un número primo es aquel que es divisible por 1 y si mismo.

En este problema definimos un anagrama primo, que significa que un número primo invertido es también un número primo. Por ejemplo el 17 después de invertir tenemos el 71. Ambos números son primos. Por lo tanto el 17 y el 71 son anagramas primos. Otros números son el 11 el 37.

Entrada

La entrada consiste de dos números a,b ($2 \leq a \leq b \leq 10,000,000$

Salida

Por cada caso de prueba escriba una línea con el número de anagramas primos que existen entre a y b inclusive.

Ejemplos de entrada

```
2 11
35 100
20 10000
```

Respuestas para el ejemplo

```
5
5
235
```

Anagrama Primo

Definición

Por definición un número primo es aquel que solo es divisible por 1 o por si mismo. Por ejemplo, el 2 es primo porque solo es divisible por 1 y por 2. Este es el único número par. Los primeros números primos son $2, 3, 5, 7, 11, 13, 17....$ El número 1 es solo divisible por si mismo, sin embargo en las definición ha sido excluído.

Podemos escribir un programa que nos permita verificar si un número es primo dividiendo todos los números comenzando por 2 hasta su raíz cuadrada (algoritmo 14). Todos los números superiores a su raíz no serán divisores.

Algoritmo 14: Verificar si un número es primo

```
1 for (i=2,√n)
2     if (n % i igual 0)
3         no es primo;
4 es primo ;
```

Todos los número no primo puede escribirse como la multiplicación de números primos, por ejemplo el 9 es el producto de 3 por tres, el 20 es el producto de 2,2 y 5.

Si tenemos que verificar cada vez si un número es primo podemos crear una lista de números utilizando el principio mencionado.

Para probar si un número es primo hay que dividir por los números primos anteriores verificando si es divisible. Para este problema utilizaremos una criba. Una criba es un tamiz que permite separar, en caso los números primos de los no primos. El algoritmo 15 muestra como desarrollar este programa.

En el algoritmo 15 se ve que el concepto es marcar todos los múltiplos de los números, de esta forma los números primos quedan sin marcar. La comparación $i * i$ es para verificar que no nos pasemos de la raíz cuadrada del tamaño máximo.

Una vez que se tiene una criba construída, solo es suficiente verificar si *criba[número]* esta en *false*. Luego invertimos el numero y hacemos la misma

Algoritmo 15: Criba de números primos

```
1 boolean criba [10000000] ;
2 for (i=2;i*i¡criba.length;i++)
3     if (criba[i] es falso)
4         for (j=i+i;j¡criba.length;j=j+i)
5             criba[j]=true ;
```

verificación y tenemos la respuesta.

6

Primos en secuencia

Como todos sabemos los números primos son aquellos que son solo divisibles por uno y por si mismos.

Los primeros 5 números primos son $2, 3, 5, 7, 11$ que para este problema los numeraremos de 0 a 4. En el problema te dan dos números A y B, en el ejemplo 0 y 4 que significa que buscamos del primer al cuarto número primo inclusive.

Entrada

Cada caso de prueba consiste en dos números A y B, ambos positivos y menores a 664,579 y donde B es mayor que A.

Salida

En la salida escriba la secuencia de números primos como se muestra en el ejemplo.

Ejemplos de entrada

```
5
0 4
3 7
1 3
1 1
1500 1501
```

Respuestas para el ejemplo

```
11 7 5 3 2 3 5 7 11
19 17 13 11 7 11 13 17 19
7 5 3 5 7
3
12577 12569 12577
```

Secuencia Prima

Definición

Por definición un número primo es aquel que solo es divisible por 1 o por si mismo. Por ejemplo, el 2 es primo porque solo es divisible por 1 y por 2. Este es el único número par. Los primeros números primos son $2, 3, 5, 7, 11, 13, 17....$ El numero 1 es solo divisible por si mismo, sin embargo en las definición ha sido excluído.

Podemos escribir un programa que nos permita verificar si un número es primo dividiendo todos los números comenzando por 2 hasta su raíz cuadrada (algoritmo 14). Todos los números superiores a su raíz no serán divisores.

Sin embargo para resolver muy fácilmente este ejercicio, es conveniente crear una criba (vea el algoritmo 15;

Una vez que tenemos la criba construída podemos construir un vector de números primos. El algoritmo 16 muestra esto.

Algoritmo 16: Copiar todos los números primos a un vector

```
1 vector [600000] ;
2 for (i=2;¡criba.length;i++)
3 |   if (criba[i] es falso)
4 |   |   vector_i = i
```

la solución del problema se logra recorriendo el vector entre los valores solicitados como se muestra en el argoritmo 17.

Algoritmo 17: Imprimir la respuesta solicitada

```
1 for (i=(A,B))
2 |   imprimir vector_i;
3 for (i=(B-1,A))
4 |   imprimir vector_i;
```

7

Manuel y las canicas

Hey, tú... Sí, ¡tú! ¿Sabes lo que es una canica? Seguro que sí. Si no lo sabes, déjame explicarte: Se trata de una pequeña esfera de vidrio o de piedra, especialmente popular entre los muchachos, ya que se utiliza para jugar. Normalmente cuando se juega con ellas se apuestan algunas y el ganador se las lleva consigo.

Manuel adora jugar con ellas, y hace poco se enteró que el supermercado cerca de su casa las está vendiendo con un gran descuento. Emocionado, pidió dinero a sus papás y fue a toda prisa. Tras elegir las que quería, se dirigió al mostrador para enterarse de una mala noticia: El descuento solo lo puede recibir si el peso total de sus canicas es exactamente X gramos.

¡Vaya! Manuel está seguro que en total sus canicas pesan más de X gramos, y el dinero que le dieron no es suficiente para comprar todas sin el descuento. Si va a comprar canicas, tendrá que dejar algunas de las que eligió en el supermercado. Ahora bien, todas las canicas que Manuel tomó son diferentes y él conoce el peso en gramos de cada una.

Tu tarea, joven informático, es realizar un programa que responda la siguiente pregunta: ¿Cuántas formas diferentes de dejar canicas hay, de modo que Manuel reciba el descuento?

Por ejemplo, si el supermercado exige que las canicas tengan un peso total de 200 gramos y Manuel tomó 5 canicas con los siguientes pesos: 100, 150, 50, 180, y 20 gr; él podría:

- Dejar las canicas que pesan 100, 150 y 50 ($180 + 20 = 200$), ó
- Dejar las canicas que pesan 100, 180 y 20 ($150 + 50 = 200$).

En total, existen dos formas diferentes de dejar canicas para recibir el descuento.

Entrada

La entrada contiene varios casos de prueba hasta fin de archivo, y la descripción de cada caso de prueba consiste de dos líneas:

- La primera línea consta de dos enteros separados por un espacio: X ($0 \leq X \leq 10000$) y N ($1 \leq N \leq 20$). X es el peso que las canicas deberían tener en total para que Manuel reciba el descuento y N es la cantidad de canicas que Manuel tomó.
- La segunda línea contiene N enteros positivos separados por espacios. Estos son los pesos (en gramos) de las canicas tomadas.

Salida

Por cada caso de prueba imprime una línea con la respuesta al problema.

Ejemplos de entrada

```
1 3
-2 1 3
5 5
1 2 3 4 5
```

Respuestas para el ejemplo

```
2
3
```

Suma de Subconjuntos

Concepto

Un subconjunto es un conjunto formado con parte de los elementos de un conjunto. Para un conjunto de tamaño n. El número total de subconjuntos es 2^n. Por ejemplo si $n = 3$ existen 8 subconjuntos.

Consideremos el conjunto {1,2,3} los 8 posibles subconjuntos son: {1 2 3 }, {1 2 }, {1 3 }, {1 }, {2 3 }, {2 }, {3 } y el conjunto vacío { }.

Para construir un programa que construya éstos subconjuntos se puede resolver en forma recursiva o como veremos con un contador binario. En el cuadro 7.1 mostramos Equivalente binario de los primeros 8 números:

Numero	Representación Binaria	Numero	Representación Binaria
0	0	1	1
2	10	3	11
4	100	5	101
6	110	7	111

Cuadro 7.1: Equivalente binario de los primeros 8 números

Primera similitud que vemos es que con tres bits podemos representar 8 números, siendo el 0 el que se utilizará para representar el conjunto vació. Viendo la relación que tiene con los del ejemplo tenemos: {1 2 3 } equivalente al 7, o sea tres unos binarios, {1 2 } equivalente al 6, o sea 110, {1 3 } es el 5 en binario 101, {1 } el 4 en binario 100, etc.

De esta muestra vemos que si incrementamos en un número binario y tomamos las posiciones de los dígitos binarios que están en uno estaremos recorriendo los subconjuntos.

El Problema

Nos piden hallar diferentes formas de escoger bolitas (sub conjunto de bolitas) tal que la suma sea un valor dado. Lo unico que hacemos es crear un contador

hasta $2^n - 1$ y para cada uno de los números, por cada uno de los bits que está en uno sumar el peso de la canica correspondiente. El algoritmo 18 muestra la solución.

Algoritmo 18: Hallar todos los subconjuntos y sumar los pesos

```
1  p vector con los pesos de las bolitas ;
2  n es el número de bolitas ;
3  k = (1 << n) − 1 el número de subconjuntos ;
4  suma = cero ;
5  x es la suma buscada ;
6  r es el número de conjuntos con la suma buscada ;
7  for (i=(1,k))
8  |  / / para cada sub conjunto sumar los pesos ;
9  |  for (j=(1,n))
10 |  |  if (1 << j&i es mayor a cero)
11 |  |  |  suma = suma + p_j ;
12 if (suma igual a x)
13 |  r = r + 1
```

8

Representación Zeckendorf

La representación Zeckendorf indica que todo número puede escribirse como la suma de números no consecutivos de Fibonacci. Por ejemplo la representación del número 100 es: $100 = 89 + 8 + 3$. Existen otras formas de representar el 100 como las suma de números de Fibonacci. Por ejemplo: $100 = 89+8+2+1$, $100 = 55+34+8+3$, pero estas representaciones no son representaciones de Zeckendorf porque uno y dos son números consecutivos de Fibonacci, así como el 34 y el 55.

Dado un número entero positivo, encuentre la representación de Zeckendorf escogiendo el numero de Fibonacci más grande en cada paso.

Entrada

La entrada consiste de un número entero positivo, menor a 10^6.

Salida

Por cada caso de entrada escriba en una línea la representación de Zeckendorf, cada número separado por un espacio.

50

Ejemplos de entrada

```
100
11
```

Respuestas para el ejemplo

```
89 8 3
8 3
```

La representación de Zeckendorf

Concepto

La representación de Zeckendorf lleva este nombre en honor al médico y matemático belga Edouard Zeckendorf.

Cuando consideramos el sistema de numeración binario vemos que todo entero positivo puede representarse de forma única como suma de potencias de 2. por ejemplo el número $17 = 2^4 + 2^0$ y lo representamos como 10001 en base 2. Con los numero de Fibonacci también podemos construir una base de numeración.

Teorema de Zeckendorf

Todo número entero positivo puede representarse de forma única como suma de números de Fibonacci distintos, de tal forma que dicha representación no contiene dos números de Fibonacci consecutivos.

Esta representación se denomina representación de Zeckendorf del número entero positivo.

Recordemos que los números de Fibonacci se obtiene de la recurrencia:

$$f(n) = \begin{cases} 0, & \text{si } x = 0. \\ 1, & \text{si } x = 1. \\ f(n-1) + f(n-2), & \text{en otros casos.} \end{cases}$$

Zeckendorf publicó su resultado en The Fibonacci Quarterly en 1972, aunque al parecer conocía esta representación desde 1939.

Para hallar la representación de Zeckendorf, tomamos el número de Fibonacci más grande de entre los que son menores que n y se lo restamos a n. Si queda cero es que el propio n era un número de Fibonacci, y si no es así repetimos el proceso las veces que sea necesario hasta que una de las restas dé cero. La correctitud de este teorema es posible demostrar por inducción haciendo uso del lema de unicidad que dice: La suma de cualquier conjunto no vacío de números de Fibonacci, no consecutivos, cuyo mayor elemento sea F_j, es estrictamente menor que el siguiente término F_{j+1}.

Para ejemplificar el proceso, recordemos que la serie de Fibonacci produce los siguientes números: $0, 1, 1, 2, 3, 5, 8, 13, 21....$ Haciendo una analogía con el sistemas de numeración binaria, vemos la representación del número 27. En números binarios se escribe como

$$27 = 2^4 + 2^3 + 2^1 + 2^0 = 11011\text{en base Fibonacci}$$

El mismo numero 27 en la representación de Zeckendrof se escribe como

$$27 = 21 + 5 + 1 = F_9 + F_6 + F_1$$

Para abreviar la representación algunos escriben un número binario 1 para indicar que este numero Fibonacci es tomado en cuenta o no. Para el 27 será 0100001001 haciendo que el número de más a la izquierda represente el Fibonacci 0.

Solución

Para escribir un algoritmo adecuado y hallar su representación Zeckendorf, tomamos en consideración los números que pueden formarse con los Fibonacci menores al F_{93} porque este es el número más grande que se puede representar en una variable 64 bits. Este programa se muestra en el algoritmo 19.

Algoritmo 19: Hallar la representación Zeckendorf de un numero

```
1 f vector con los números Fibonacci ;
2 n es el número a representar ;
3 while (n mayor a cero)
4 |   Buscar el número f_i más próximo a n ;
5 |   Poner el resultado en x ;
6 |   n= n-x ;
7 |   imprimir x ;
```

Variantes

Para este problema se pueden plantear varias variantes o problemas nuevos basados en las representaciones de Zeckendorf. Por ejemplo dados dos números en su representación Zeckendorf como se hace la multiplicación y/o suma de los mismos?

Siendo que la relación entre millas y kilómetros es un número cercano a la razón de los números Fibonacci, como se puede aproximar la conversión entre kilómetros y millas utilizando la notación de Zeckendorf?.

Como se mostró en la ecuación 2.3 la razón de los con la que los números de Fibonacci es de 1.618 y la relación entre kilometras y millas es de 1.66. Una milla es 1.66 kilómetros. Para este segundo reto existen muchas páginas de internet que tienen ejemplos, pero con solo el propósito de hacer un ejemplo mostramos:

Para convertir 72 millas a kilómetros primero escribimos las representación Zeckendorf

$$72 = 55 + 13 + 3 + 1$$

Luego substituimos cada número por su Fibonacci inmediato superior $89 + 21 + 5 + 2 = 117$

de donde 72 millas serán, aproximadamente 117 kilómetros.

Para convertir de kilómetros a millas hacemos el proceso inverso.

Índice de figuras

Índice de cuadros

Índice de algoritmos