Olimpiada de Informática

—

—

II

Índice general

Parte I

Elementos básicos

Parte II

Búsquedas

Capítulo 1

Búsqueda exhaustiva
Fuerza bruta

Muchas veces en nuestra vida hemos tenido que buscar algo, una foto en nuestra galería del teléfono, una palabra en el diccionario, una carta dentro de un mazo, etc. Y probablemente, con la experiencia, hemos aprendido algunas intuiciones sobre como buscar cosas, en esta parte trabajaremos un poco más en desarrollar esta intuición e ideas.

En la olimpiada de informática también debemos buscar cosas. Ya sea encontrar la solución al problema o solo utilizar una búsqueda como paso intermedio, ser buenos haciendo búsquedas nos abrirá la puerta a muchos problemas y técnicas.

Para comenzar con entendiendo las búsquedas comenzaremos con la búsqueda más sencilla, la búsqueda exhaustiva, también conocida como fuerza bruta.

Lo primero que se requiere para poder buscar es definir el espacio de búsqueda, ¿dónde podría estar lo que queremos encontrar? De la respuesta dependerá como haremos la búsqueda. Por ejemplo, si buscamos la posición de un valor en un arreglo, el espacio de búsqueda es el arreglo.

Una vez que definamos donde podría estar la respuesta, lo que hacemos con la exhaustiva es explorar absolutamente todo el espacio de búsqueda, todos los candidatos que podrían ser lo que buscamos hasta dar con la respuesta. Por esto se le conoce como fuerza bruta, porque aprovecha el poder computacional para procesar todo hasta dar con la respuesta.

Se te enseñara las búsquedas exhaustivas más comunes, así como una forma general. En concreto, los espacios de búsquedas que usaremos son:

- Espacio lineal
- Parejas de elementos
- Ordenes (Permutaciones)
- Subcojuntos

Y terminaremos este capitulo aprendiendo como hacer búsqueda exhaustiva de forma general usando recursión e iteración.

1.1. Búsqueda lineal

La búsqueda lineal es la más sencilla de las búsquedas que hay. ¿Qué es lo que harías si te pido que de una pila de exámenes encuentres el tuyo? Lo que probablemente hagas es una revisar uno por uno, checar de arriba hacia abajo hasta que encuentres el examen con tu nombre en él.

Básicamente, esta idea es la búsqueda lineal, ir revisando de uno por uno toda una lista de candidatos hasta encontrar al que estas buscando o hasta que hayas revisado todos los candidatos.

Entonces, los códigos de búsqueda lineal casi siempre tendrán la siguiente estructura:

```
Itera por cada candidato {
   Si el candidato es lo que buscamos {
      Respuesta = candidato;
      Detener ciclo /* esto es opcional, depende si hay
          varios valores que queramos encontrar. */
   }
}
```

Veamos cómo usar esta técnica para resolver un problema.

Ejemplo: Encontrar posición en un arreglo

Supongamos que queremos tenemos un arreglo A de enteros distintos y este tiene N elementos en él. Nosotros querremos hacer un código que imprima la posición del arreglo que valga K. O si este valor no existe, que imprima -1.

Límites

- $1 \leq N \leq 10^5$
- $1 \leq K, A[i] \leq 10^9$

ENLACE: TODO

Solución

(Recuerda intentar el problema antes de leer la solución)

Lo que este problema nos pide en realidad es buscar dentro del arreglo por el índice del elemento K.

Lo que haremos es revisar todas las posiciones del arreglo hasta encontrar aquella que valga K, si no la encontramos imprimimos -1.

Código

```
int A[100050];
int N, K;
int main () {
   ios_base::sync_with_stdio(0); cin.tie(0);
   cin >> N;
   for (int i=0; i <N; i++){
      cin >> A[i];
   }
   cin >> K;
   int respuesta = -1;
   for (int i =0; i < N; i++) {
      if (A[i]==K) {
         respuesta = i;
         break;
      }
```

```
    }
    cout << respuesta;
}
```

1.1.1. Complejidad

Una pregunta que te has de hacer es: ¿cuál es la complejidad de esta técnica? Y la respuesta es sencilla, en el peor de los casos tenemos que revisar a todos los candidatos, digamos que la cantidad de ellos es iguala a N, entonces la complejidad es $O(N)$.

Ejemplo: Contar formas de dividir

Veamos otro problema de búsqueda lineal.

Descripción

Carlos quiere armar una fiesta, y como le gusta ser un buen anfitrión compro N regalos para sus invitados. Ahora, Carlos quiere darle la misma cantidad de regalos a cada uno de sus invitados sin que sobre ningún regalo no repartido. Como Carlos le gusta contar, ahora se pregunta: ¿Cuántas cantidades diferentes de invitados puede tener?

Entrada

Un entero N, indicando cuantos regalos compró Carlos.

Salida

La cantidad de posibles números de invitados para la fiesta.

Casos ejemplo

Entrada	Salida
12	6
7	1
4	3

Límites

- $1 \leq N \leq 10^5$

Solución

Es fácil ver que el problema en realidad pregunta: ¿Cuántos divisores positivos tiene N?

(Nota: un divisor de N es un número que divide a N sin decimales).

Encontremos todos los divisores de N. Estos se encontrarán entre 1 y N, por lo que podemos iterar i por todo este rango revisando si i es divisor de N.

Código

```
respuesta = 0;
for (int i =1; i <= N; i++) {
   if (N%i==0) {
      respuesta++;
   }
}
cout << respuesta;
```

Problema 1.1 *Crea un problema que cuente cuantas veces aparece el valor K en un arreglo. ($1 \leq N \leq 10^5$)*
(omegaup.com/arena/problem/frecuencia-de-k)

1.2. Búsqueda lineal con función de validación

Hasta ahorita hemos visto problemas donde revisar si un candidato era la respuesta o no bastaba con un simple condicional, pero este no siempre es el caso.

Varías veces, para revisar si un valor es solución a nuestro problema, vamos a tener que necesitar un poco más de código e ideas. Veamos un problema de este estilo.

Ejemplo: Dividiendo listones

Descripción

Karel tiene N listones de distintas longitudes enteras. Karel quiere hacer pulseras con ellos, por lo que tomará cada uno de los listones y los cortará para que las pulseras usen segmentos del mismo tamaño.

A Karel le gustan los enteros, entonces la longitud de los segmentos también ha de serla. Además, Karel no quiere que sobre listón sin usar ¿Cuántos diferentes tamaños de segmento se pueden elegir?

Entrada

Un entero N, indicando la cantidad de listones En la siguiente línea, N enteros indicando las longitudes de los listones. Llamemos $A[i]$ a la longitud del listón i.

Salida

La cantidad de opciones para el tamaño de los segmentos

Caso ejemplo

Entrada	Salida	Expliación
5 10 30 20	3	Las longitudes pueden ser 1, 2 y 5

Límites

- $1 \leq N \leq 100$
- $1 \leq A[i] \leq 5000$

Código

Encontremos con búsqueda lineal todos los tamaños de segmento que cumplen y contemos cuantos son.

Primero veamos que los tamaños de listón deben estar entre 1 y 5000. Más concreto, entre 1 y $min(A[1], A[2], \cdots, A[N])$. Esto es porque el tamaño del segmento debe ser entero, debe ser por lo menos 1 (ya que un segmento de tamaño 0 o menor no tiene sentido para este problema) y no puede ser más largo que el listón más corto.

Ya hemos visto como se ve una búsqueda lineal y la que usaremos en este caso sería:

```
cin >> N;
for (int i=0; i< N; i++) {
   cin >> A[i];
}

int minA=A[0];
for (int i=1; i < N; i++) {
   minA=min(minA, A[i]); /* encuentra el liston mas
```

```
   pequeno. */
}
respuesta = 0;
for (int s =1; s <= minA; s++) {
   if (es s es un tamano de segmento valido) {
      respuesta++;
   }
}
cout << respuesta
```

Pero el reto ahora es el chequeo de "es s es un segmento de tamaño valido".

Para esto necesitamos un poco más de trabajo. Veamos un solo listón. Si queremos cortarlo en segmentos de tamaño s sin que sobre, ¿qué tiene que cumplir s con relación al listón? Así es, que es, que s divida a la longitud del listón. Y podemos ver que s tiene que cumplir esto para todos los listones

Entonces, para ver que s sea una opción válida, hay que ver que s divida a todos los enteros en la lista de listones.

Para lograr esto, creemos una función booleana que se encargue de validar s.

```
bool validar (int s) {
   bool respuesta = true;
   for (int i=0; i< N; i++) {
      if (A[i]%s!=0){
         respuesta = false;
         break;
      }
   }
```

```
    return respuesta;
}
```

Entonces con esta función obtenemos que el código de la búsqueda lineal ahora es:

```
respuesta = 0;
for (int s =1; s <= minA; s++) {
    if (validar(s)) {
        respuesta++;
    }
}
cout << respuesta
```

Y con esto logramos completar el problema.

1.2.1. Complejidad

La búsqueda lineal la hacemos sobre el valor de A, pero, además, por cada iteración de la búsqueda lineal, hacemos un ciclo que revisa la condición para que s sea contada.

Entonces, la complejidad nos queda como: $O(Busqueda \times Validar) = O(AN)$.

Como $A \leq 5000$ y $N \leq 100$. Nos queda que $AN \leq 5 \times 10^5$, lo cual corre en menos de un segundo.

1.3. Parejas de elementos

En muchos problemas nos pedirán que encontremos o contemos la cantidad de pares que cumplan alguna condición, o nosotros convertiremos a un problema de este estilo. Para este tipo de problemas será útil conocer como hacer una búsqueda completa que revise todas las posibles parejas de elementos.

Como antes, veamos un problema que puede ser resuelto con esto.

Ejemplo: Pares de suma K

Fernando necesita K tornillos de la ferretería. Sin embargo, la vida no siempre es fácil y la ferretería no vende exactamente K tornillos.

Sin embargo, venden N cajas de tornillos cada una con C_i tornillos dentro.

Como Fernando tiene una obsesión con no desperdiciar, él solo comprara las cajas de forma que traigan juntas exactamente K tornillos. Además, odia las bolsas de un solo uso que dan en la ferretería por lo que solo comprará dos cajas, una por cada mano.

Entonces, dado el tamaño de las cajas, determina si Fernando puede traer consigo exactamente K tornillos.

Entrada

Dos enteros, N y K, representando cuantas cajas hay y cuantos tornillos se requieren.

En la siguiente línea vendrán N enteros separados por espacios, indicando la cantidad de tornillos en cada caja.

Salida

Deberás imprimir "SI" en caso de que Fernando pueda obtener K tornillos con sus reglas, o "NO" si es imposible.

Casos ejemplo

Entrada	Salida	Expliación
4 6 3 1 8 5	SI	Usa las cajas con 1 y 5 tornillos.
5 10 3 1 8 5 12	NO	

Límites

- $1 \leq N \leq 1000$
- $1 \leq K \leq 10^9$
- $1 \leq C_i \leq 10^9$
- $C_i \neq K$

Enlace: [TODO]

Solución

Lo que nos pregunta el problema es: ¿Existe un par de cajas tal qué sumen K?

Para determinar si existe dicha pareja, lo que haremos será buscar entre todas las parejas de cajas aquella que sume K. Es decir, buscaremos completamente todas las parejas posibles.

Para iterar por todas las parejas lo que haremos es primero definir una

pareja como dos índices (i, j), tal que $0 \leq i < j < N$. Como queremos iterar por todos los posibles pares, primero iteraremos por todos los valores de i. Y para cada i, iteraremos por todas las j con las que se puede emparejar. El código se ve como:

```
for (int i =0; i < N; i++) {
   for (int j=i+1; j< N; j++) {
      cout << i<<" "<<j<< "\n";/* imprimimos
      cada par */
   }
}
```

Y ahora que sabemos iterar por todos los pares, lo utilizamos para revisar si existe un par que sume K.

```
for (int i =0; i < N; i++) {
   for (int j=i+1; j< N; j++) {
      if (Caja[i]+Caja[j] == K) {
         cout << "SI";
         exit(0); /* Termina el programa,
         encontramos la respuesta */
      }
   }
}
cout << "NO";
```

Entonces, son estos dos ciclos for nos permiten iterar por toda pareja de elementos en un arreglo. Esta es una herramienta bastante útil para resolver muchos problemas y subtareas.

1.3.1. Complejidad

Bien, ahora hablemos de la complejidad de esta técnica. La complejidad es $O(N^2)$. Esto es porque la cantidad de parejas con N elementos crece en$O(N^2)$.

Pero incluso si no conocemos como crecen las parejas, podemos ver que este ciclo para $i = 0$, itera por $N-1$ valores de j; para $i = 1$, iteramos por $N-2$ valores de j; para $i = 2$, iteramos por $N-3$ valores de j, y así sucesivamente. De forma que hacemos $(N-1)+(N-2)+\ldots+1+0$ iteraciones de j. Entonces hacemos $1+2+3+\ldots+(N-1)$ iteraciones.

Usando la formula se suma de gauss[1] obtenemos que:

$$1 + 2 + 3 + \ldots + (N-1) = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2}$$

Entonces, la complejidad queda como $O(\frac{N^2}{2} - \frac{N}{2}) = O(N^2)$

Por lo tanto, iterar por todos los pares de un arreglo es una técnica de complejidad cuadrada, perfecta para límites hasta $\sim 10^4$.

Problema 1.2 *Una inversión en un arreglo es una pareja de números i y j que cumplen: $i < j$ y $A[i] > A[j]$, es decir, estos dos números están desordenados con relación entre ellos.*

Dado un arreglo de N enteros, determina cuantas inversiones tiene. ($1 \leq N \leq 1000$).

(TODO)

[1] Formula de gauss para sumar los primeros N naturales: $1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2}$

Problema 1.3 *Búsqueda lineal encuentra un solo elemento, aquí aprendimos a buscar una pareja, dos elementos. Crea el código para buscar una tercia de elementos en un arreglo que sume K.*

No olvides analizar la complejidad.
(TODO).

1.4. Ordenes o permutaciones

En algunos problemas nos pedirán que encontremos un ordenamiento.

TODO HACER ESTO

1.5. Subconjuntos

1.5.1. Conjuntos y subconjuntos

Un conjunto no es nada mas que una colección de objetos. Por ejemplo, si alguien trae en su mochila un plátano, manzana y naranja, este puede decir que trae un conjunto de frutas compuesto por un plátano, manzana y naranja.

En matemáticas e informática, nosotros estamos trabajando todo el rato con los conjuntos. Ya sean el conjunto de datos de entrada, o los números enteros, tener noción de conjuntos es un requerimiento para el éxito en la olimpiada.

Veamos un poco de notación. Para escribir el conjunto A de números esta conformado por el 3, 5 y 9 escribimos lo siguiente:

$$A = \{3, 5, 9\}$$

TODO: CORREGIR ESTO

Ejemplo: Enlistar subconjuntos

Veamos como hacer para visitar todos los subconjuntos, lo cuál será necesario para problemas donde nos preguntan por ellos.

Para esto, primero aprendamos a resolver el problema que trata de mostrar los subconjuntos:

Problema:

Karel tiene un conjunto formado por las primeras N letras del alfabeto. Ahora Karel que imprimas todos los subconjuntos, uno por cada línea. Puedes imprimirlos en cualquier orden.

Cada subconjunto es representado por las letras en el de la A a la Z. El subconjunto vacío será representado por un asterisco '*'.

Entrada
Un entero N, indicando cuantas letras hay en el conjunto.

Salida
Todos los subconjuntos

Ejemplos

Entrada	**Salida**
2	AB A B *

Entrada	Salida
3	ABC AB AC A BC B C *

Límites

- $1 \leq N \leq 20$

Solución

Entonces, podemos resolver el problema si obtenemos todos los subconjuntos posibles. Para esto, haremos un algoritmo que construya todos los posibles subconjuntos.

Esto se puede hacer de dos formas principales, una iterativa y otra recursiva. Comencemos viendo la forma con recursión.

Subconjuntos usando recursión

Digamos que $S(conjunto)$ sean todos los subconjuntos del *conjunto*. Por ejemplo:

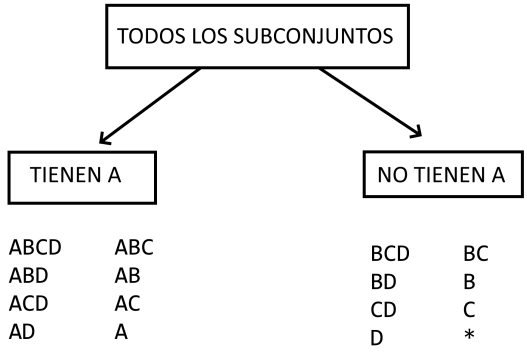
$$S(\{A, B, C\}) =$$

$$\{\{A, B, C\}, \{A, B\}, \{A, C\}, \{A\}, \{B, C\}, \{B\}, \{C\}, \emptyset\}$$

O usando la misma notación que la salida del problema (la que usaremos a partir de aquí):

$$S(ABC) = \{ABC, AB, AC, A, BC, B, C, *\}$$

Veamos un poco como se comporta S, por ejemplo para $N = 4$, en total tenemos 16 subconjuntos los cuales podemos dividir en dos mitades con 8 cada una, los que tienen la A y los que no tienen la A.



Ahora veamos que para los dos grupos, tenemos todos los 8 subconjuntos para BCD, y lo único que los diferencia es si tienen A al inicio o no. Si supiéramos cuales son esos subconjuntos para las ultimas 3 letras, podríamos perfectamente construir $S(ABCD)$,

En concreto, podemos ver que $S(ABCD)$ es igual a $S(BCD)$ con A al inicio y $S(BCD)$ sin nada extra.

Si lo quieres ver en formula sería similar a $S(ABCD) = A \to S(BCD) \cup S(BCD)$, donde $A \to S(BCD)$ significa agregar A a todos los subconjuntos en $S(BCD)$.

Y podemos hacer lo mismo, ver que $S(BCD)$ es otra vez: los subconjuntos de CD con B y sin B.

Y de aquí obtenemos nuestro comportamiento recursivo.

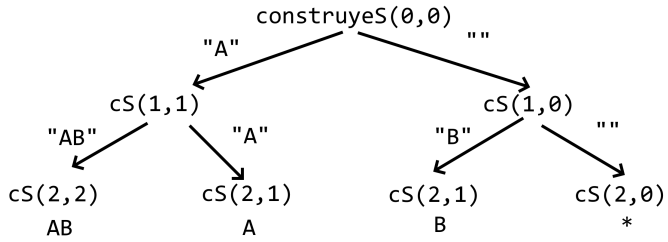
Ahora que vemos la recursión, veamos una forma de implementar todo esto para resolver el problema.

Crearemos un metodo construyeSubconjuntos(int pos, int previo) que constuya los subconjuntos usando las letras desde pos hasta $N - 1$.

```
#include <iostream>
using namespace std;
int N;
char subconjunto[25];

// pos implica que letra estamos eligiendo si esta o no
    A=0, B=1, C=2, ...
// para N=6, pos = 2, es equivalente a S(CDEF)
// previo indica cuantas letras se han agregaron a la
    construccion.
void construyeSubconjuntos(int pos, int previo) {
   if (pos == N) {
      //Ya no hay mas letras que decidir
      //imprime el subconjunto construido.
      if (previo==0) {
         cout <<"*\n";
      } else {
         cout << subconjunto<<"\n";
      }
      return;
   }
   //agrega la letra pos a la construccion
   subconjunto[previo]=pos+'A';
   construyeSubconjuntos(pos+1, previo+1);

   // Quita la letra pos de la construccion
   subconjunto[previo]='\0';
   construyeSubconjuntos(pos+1, previo);

}

int main () {
   cin >> N;
```

```
    construyeSubconjuntos(0,0);
    return 0;
}
```

De forma que el árbol de la recursión para construyeSubconjuntos(0,0) con $N = 2$ se ve:



1.5.2. Complejidad

a complejidad de este código es igual a la cantidad de subconjuntos que se tienen con N elementos.

La primera forma de hacerlo es darnos cuenta que cada elemento tiene dos opciones, estar o no estar, por principio multiplicativo vemos que se multiplica por dos tantas veces como elementos tengamos. En total es 2^N

Esto es observable si recordamos el diagrama de las llamadas recursivas.

Vemos que por cada elemento, se divide en dos, duplicando la cantidad de llamadas en el proceso. En el primer nivel con pos=0, tenemos solo una llamada, con pos=1 tenemos dos, pos=3 obtenemos cuatro y así

sucesivamente.

Por lo tanto, la complejidad es $O(2^N)$, exponencial.

Una vez comprendido esto, pasemos a utilizar esto para resolver problemas.

Ejemplo Subconjunto de Suma K

Fernando ha llegado a la ferretería con su objetivo frecuente de comprar K tornillos. Sin embargo, la ferretería no siempre vende cajas con exactamente K tornillos dentro.

La ferretería tiene N cajas diferentes en venta, cada uno con C_i tornillos dentro.

Fernando quiere comprar unas cuantas cajas y obtener **exactamente** K tornillos. Por fortuna, esta vez trajo una bolsa y podrá comprar cuantas cajas le sea necesario.

Conociendo las cajas que venden en la Ferretería, determina si Fernando puede comprar los K.

Entrada
Dos enteros N y K, la cantidad de cajas en la Ferretería y cuantos tornillos quiere Fernando.

En la siguiente línea vendrán la cantidad de tornillos en cada caja, los valores C_i, separados por espacios.

Salida
Debes imprimir "SI.^en caso de que Fernando pueda comprar exactamente K tornillos. Caso contrario, imprime "NO".

Ejemplos

Entrada	Salida	Expliación
5 10 2 4 5 3 9	SI	Compra la primera, tercera y cuarta caja.
5 12 4 5 2 11 3	NO	

Límites

- $1 \leq N \leq 20$
- $1 \leq C_i \leq 10^9$
- $1 \leq K \leq 10^9$

Solución

Como siempre, resumamos el problema en menos palabras. En corto, nos preguntan si existe un subconjunto de cajas tal que la suma de sus valores sea exactamente K.

Y como es propio de todos los subtemas de búsqueda completa, veamos todas las posibles soluciones, es decir todos los subconjuntos, hasta que encontremos el que cumpla la condición o nos quedemos sin opciones.

Como ya vimos la forma de iterar por todos los subconjuntos en el ejemplo 3.1, utilicemos esas ideas para resolver este problema.

Lo que podemos hacer es ir construyendo todos los subconjuntos, esta vez en vez de representarlos como una cadena, lo representaremos como su suma. Esto en código se ve:

```
int Cajas[25];
int N;
long long K;
// Aqui, llevamos la suma del subconjunto construido
void buscaSubconjunto(int pos, long long suma) {
   if (pos==N) {
      if (suma == K) {
         /* Encontramos un subconjunto con suma K.
         Imprimmos SI y terminemos el programa */
         cout << "SI";
         exit(0);
      }
      return;
   }
   buscaSubconjunto(pos+1, suma+Cajas[pos]);//Incluyamos
       pos en el subconjunto
   buscaSubconjunto(pos+1, suma);//Excluyamos pos del
       subconjunto
}

int main () {
   ios_base::sync_with_stdio(0); cin.tie(0);
   cin >> N >>K;
   for (int i=0; i < N; i++) {
      cin >> Cajas[i];
   }
   buscaSubconjunto(0,0);
   //Si llegamos aqui es porque nunca encontramos la
       respuesta.
   cout << "NO";
   return 0;
}
```

1.6. Forma recursiva

Ahora se verá como crear cualquier búsqueda exhaustiva con recursión, lo cual nos va a servir para cualquier fuerza bruta que no hayamos visto en la lista anterior. Es la forma general de cualquier exhaustiva.

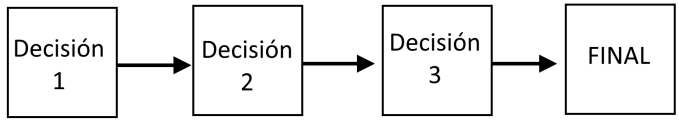
Para resolver esto, imaginaremos que lo que queremos buscar es una una secuencia de valores que cumplan algo.

Si nos fijamos, todos los ejemplos anteriores fueron eso, buscar una secuencia de valores:

Por ejemplo, en la búsqueda lineal buscábamos una secuencia de un solo número, por ejemplo [4] que representa el valor de la raíz cuadrada de 16. Mientras que en las parejas de elementos queríamos una secuencia de dos números que representen la pareja, por ejemplo $[1, 6]$, los indices de un arreglo.

En los ordenes, la permutación tal cual es la secuencia, y en los subconjuntos teníamos una cadena de tamaño n con valores de 0 o 1 representando si un elemento esta o no.

Ahora, la pregunta es ¿cómo buscamos las secuencias de números? para una secuencia lo que podemos imaginar es que tomamos la decisión de cuanto vale el primer elemento de la secuencia, luego elegimos cuanto vale el segundo elemento, luego el tercero, etc. Formando una cadena de decisiones hasta que lleguemos al final de la secuencia.



Y la idea de esta búsqueda completa es revisar todas las formas posibles de tomar esta serie de decisiones para encontrar todas las cadenas.

Para esto hacer esto, cada que tengamos que tomar una decisión, probamos todas las opciones disponibles para ver que producen.

Supongamos que estamos buscando la solución y tenemos una decisión ante nosotros con dos opciones. Lo que la búsqueda exhaustiva hace es ver "¿qué pasa si elijo la primera opción?"para luego revisar "¿Y qué sucede con la segunda opción?". Si después hay más decisiones, también explorara todas las opciones que aporten.

De forma que la búsqueda se ve aproximadamente de la siguiente forma:

```
busqueda(decision, solucion) {
   if (decision es el final) {
      revisar solucion contruida;
      return;
   }
   //Revisa cada opcion de esta decision:
   for (int i=0; i < decision.opciones; i++) {
      busqueda(siguiente_decision, solucion+
          decision.opcion[i] );
   }
}
```

Probablemente reconozcas este código de la forma recursiva de iterar por todos los subconjuntos, esto es porque utilizamos esta técnica para resolver aquella fuerza bruta.

Como es costumbre, veamos ejemplos para entender esto:

Ejemplo: Pares de suma K

Dado un arreglo A de N enteros, determina si existe un par i y j $(i < j)$ tal que $A[i] + A[j] == K$.

Ejemplo

Entrada	Salida
5 8 3 1 2 5 9	SI
5 10 3 4 2 12 9	NO

Límites

- $1 \leq N \leq 10^3$
- $1 \leq A[i], K \leq 10^9$

Solución

Este problema lo vimos en la sección de iterar por pares,lo que debíamos hacer era revisar todas las posibles parejas y ver si existía aquella que sume K.

```
bool respuesta=false;
for (int i=0; i < N;i++) {
```

```
    for (int j=i+1; j < N; j++) {
       if (A[i]+A[j]==K) {
          respuesta=true;
       }
    }
 }
```

Pero ahora pensemos un poco a más profundidad que hace este código.

Primero va revisando todas las opciones de i. Y para cada opción, evalúa todas las opciones para j.

Tiene que elegir cuanto valen dos valores, probará todos los valores para i con el primer ciclo, revisando un valor por cada iteración. Y para cada valor de i probará todos los valores de j posibles.

Esto se puede ver también con una recursión:

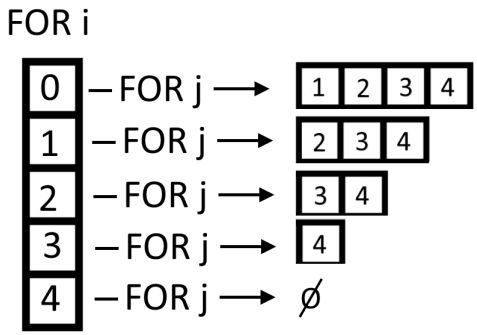
```
int solucion[3];
bool respuesta=false;
void buscar(int decision) {
   if (decision==3) {
      //Ya tomamos las dos decisiones, cuanto vale i, cuanto
          vale j. Revisemos esta solucion
      if (A[solucion[1]]+A[solucion[2]]==K) {
         respuesta=true;
      }
      return;
   }
   if (decision == 1) {
      //toca decidir cuanto vale i. Como esto es busqueda
          completa, revisaremos cada opcion.
      for (int i=0;i < N; i++) {
         solucion[decision]=i;//agregar i a la solucion
         buscar(decision+1);
         solucion[decision]=-1;//quitar i de la solucion
      }
   } else {
      //toca decidir cuanto valdra j. Probemos todos los
          valores
      for (int j= solucion[1]+1; j< N;j++) {
         solucion[decision]=i;//agregar j a la solucion
         buscar(decision+1);
         solucion[decision]=i;//quitar j.
      }
   }
}


int main() {
   [...]
```

```
    buscar(1);
    cout << "NO";
    return 0;
}
```

Estos dos códigos siguen la misma idea de obtener los pares fijando primero un valor de i, y luego fijando un valor para j; solo lo realizan de una forma diferente.

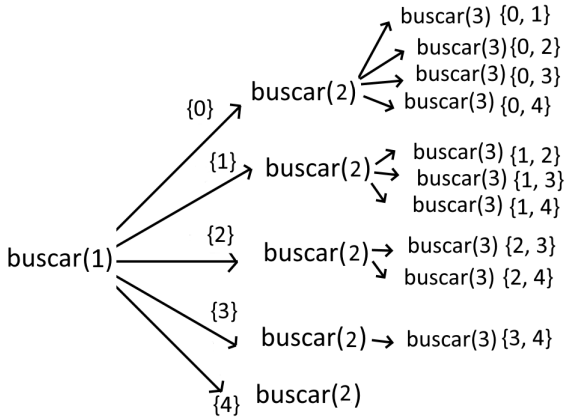
Consejo: Entiende porque los dos códigos resuelven el problema con la misma idea antes de continuar, prueba a ejecutar ambos a mano con lápiz y papel.

Básicamente el primer código hace lo siguiente para explorar todos los pares:



Donde para cada elemento de la lista de valores de i, prueba los valores de j.

Mientras que el segundo hace recursión para obtener todos los pares posibles.



Ambos son búsquedas exhaustivas que exploran todas las formas de tomar las decisiones de elegir i y luego j.

Veamos otro ejemplo:

Ejemplo: Dividir en tres grupos

Una clase se ha juntado para jugar LaserTag. Por el diseño del lugar donde van a jugar han decidido formar tres equipos, pero quieren que los jugadores sean repartidos justamente.

La clase consiste de n estudiantes. El estudiante i tiene a_i habilidad para el LaserTag.

Los equipos están repartidos de forma justa si la suma de la habilidad de sus integrantes es igual para los tres equipos y nadie se queda sin equipo.

Determina si se puede formar los equipos de forma justa y si sí, deter-

mina la repartición.

Entrada
Un entero n, indicando la cantidad de estudiantes en la clase.

La segunda línea tendrá n enteros $a_1, a_2, \ldots, a_n$, donde a_i es la habilidad del estudiante i.

Salida
Si no se puede hacer la repartición imprime `NO`.

En caso de que se pueda, en la primera línea imprime `SI`.

En la segunda línea imprime n enteros. El i-ésimo entero sera 1, 2 o 3 dependiendo de a que equipo va el estudiante i.

Si hay varias soluciones, se acepta cualquiera.

Ejemplo

Entrada	**Salida**	**Expliación**
6 3 1 6 2 4 2	SI 1 1 3 2 2 1	El primer equipo tiene una habilidad de $3 + 1 + 2$. El segundo tiene habilidad de $2 + 4$. Y el tercer equipo tiene un integrante de 6.
6 2 1 5 2 3 4	NO	No se puede repartir a los estudiantes justamente.

Límites

- $1 \leq N \leq 15$

- $1 \leq a_i \leq 10^6$

ENLACE: TODO

Solución

Como siempre, antes de leer la solución te invitamos a que trates de resolver el problema por un rato.

Entonces, podemos pensarlo como que tenemos N decisiones, donde cada decisión es ¿A que equipo envío al estudiante i?

Y lo podemos imaginar como que los estudiantes se forman delante de nosotros y les vamos diciendo: ."Equipo 1, equipo 2, equipo 1, equipo 3, ... ".

Entonces, podemos hacer un código que haga eso, que vaya tomando las decisiones de enviar a cada estudiante al equipo 1, 2 o 3.

Para esto crearemos la función repartir(int c, int equipo1, int equipo2, int equipo3) que se encargara de decidir a cual equipo mandar al estudiante c. Y llevaremos cuenta de cuanta habilidad lleva cada equipo hasta ahora.

Además usaremos un arreglo `reparticion[]` para llevar cuenta de a que equipo mandamos cada estudiante.

```
int n;
int a[16];
int reparticion[16];
void repartir(int c, int equipo1, int equipo2, int equipo3)
{
   //Mandar el estudiante c al equipo1:
   reparticion[c]=1;
   repartir(c+1, equipo1+a[c], equipo2, equipo3);

   //Mandar el estudiante c al equipo2:
   reparticion[c]=2;
   repartir(c+1, equipo1, equipo2+a[c], equipo3);

   //Mandar el estudiante c al equipo2:
   reparticion[c]=3;
   repartir(c+1, equipo1, equipo2, equipo3+a[c]);
}
```

Ahora, le falta la condición de paro. Esto será en el momento que ya no tengamos estudiantes que repartir, cuando $c = n$.

Además, aprovecharemos allí para validar que la repartición sea justa. Si lo es, imprimiremos esta construcción como respuesta y terminaremos el programa.

```
int n;
int a[16];
int reparticion[16];
void repartir(int c, int equipo1, int equipo2, int equipo3)
{
   if (c==n) {
      if (equipo1==equipo2 && equipo1==equipo3) {
         cout << "SI\n";
         for (int i =0;i<n; i++)
         cout << reparticion[i]<<" ";
         exit(0);
      }
      return;
   }
   //Mandar el estudiante c al equipo1:
   reparticion[c]=1;
   repartir(c+1, equipo1+a[c], equipo2, equipo3);
   //Mandar el estudiante c al equipo2:
   reparticion[c]=2;
   repartir(c+1, equipo1, equipo2+a[c], equipo3);
   //Mandar el estudiante c al equipo2:
   reparticion[c]=3;
   repartir(c+1, equipo1, equipo2, equipo3+a[c]);
}
int main() {
   ios_base::sync_with_stdio(0); cin.tie(0);
   cin >> n;
   for (int i =0; i< n; i++)
      cin >> a[i];
   repartir(0, 0, 0, 0);
   cout << "NO";
}
```

1.6.1. Complejidad

La complejidad de una búsqueda exhaustiva varia mucho dependiendo de que tipo de solución se esta buscando, en el primer ejemplo terminamos con una complejidad de $O(N^2)$.

Pero la complejidad del ejemplo anterior es $O(3^N)$. Esto es porque la complejidad de la búsqueda exhaustiva es $O(candidatos)$ y en este caso, las formas de repartir se multiplican por tres por cada estudiante que se agregue a la clase, ya que tenemos todas las opciones de equipos anteriores, pero unas con el nuevo en el 1, otros con el equipo 2, y otra vez con el estudiante en el 3.

Problema 1.4 *Prueba a programar la búsqueda lineal con está técnica.*

(omegaup.com/arena/problem/TODO (Busca el valor en arreglo))

Problema 1.5 *Intenta programar un código que genere todos los ordenes posibles con recursión.*

(omegaup.com/arena/problem/TODO (Busca el valor en arreglo))

Problemas de práctica

Problema 1.6 *TODO Multiplos de cinco*
(omegaup.com/arena/problem/multiplos-cinco)

Problema 1.7 *TODO: Divisores del entero*
(omegaup.com/arena/problem/divisores-entero)

Problema 1.8 *A la suma de digitos*
(omegaup.com/arena/problem/m-suma-digitos)

Problema 1.9 *Escalera de fer*
(TODO)

Problema 1.10 *Bicicleta de Karel I*
(omegaup.com/arena/problem/bicicleta-de-karel-i)

Problema 1.11 *Contar capicúas*
(omegaup.com/arena/problem/Contar-capicuas)

Problema 1.12 *Cuenta primos*
(omegaup.com/arena/problem/Cuenta-primos)

Problema 1.13 *TODO Puntuación de la ferretería*
(omegaup.com/arena/problem/contar-sumas-pares)

Problema 1.14 *TODO Números de moda*
(omegaup.com/arena/problem/moda-fibonacci)

Problema 1.15 *TODO Terreno más valios*
(omegaup.com/arena/problem/TODO terreno-valioso)

Problema 1.16 *TODO Suma modular*
(omegaup.com/arena/problem/TODO suma-modulo-k)

Problema 1.17 *Warel roba diamantes*
(omegaup.com/arena/problem/warel-roba-diamantes)

Problema 1.18 *Imprime números binarios*
(omegaup.com/arena/problem/Imprime-binario)

Problema 1.19 *Invirtiendo una matriz*
(omegaup.com/arena/problem/TODO reverse-matrix)

Problema 1.20 *Cuenta cuantas sumas diferentes distintas hay en los subconjuntos de un arreglo.*

Contando LaserTag justos
(TODO)

Problema 1.21 *Subiendo la torre*
(TODO)

Problema 1.22 *Mapas*
(omegaup.com/arena/problem/OMI-2020-Mapas)

Problema 1.23 *TODO FIX THIS: Dado un arreglo de N elementos, responde Q queries. $N \leq 20$ $Q \leq 10^5$*

Cada query serán dos elementos s y t, determina si existe un subconjunto de tamaño t de suma s. $s \leq 1000$

Preguntas sumisticas
(TODO)

Capítulo 2

Búsqueda con podas Backtracking

Sigamos con una búsqueda que evita probar absolutamente todo el espacio de búsqueda, en este caso es la búsqueda de vuelta a atrás, o backtracking en inglés.

Esta búsqueda es similar a la búsqueda exhaustiva, pero estando atentos a no explorar candidatos en los que nos demos cuenta que no encontraremos lo que buscamos.

Como acostumbramos veamos un ejemplo.

Ejemplo: Subconjunto de suma K

Dada una lista de N enteros positivos, determina si existe un subconjunto que sus elementos sumen K.

Entrada
Dos enteros N y K — La cantidad de enteros y la suma que queremos.

En la segunda línea hay N enteros $A_1, A_2, \ldots A_N$ — Los valores de la lista. **Salida**
Imprime `SI` en caso de que exista un subconjunto de suma K, o imprime `NO` en caso contrario.

Ejemplos

Entrada	Salida	Expliación
4 12 5 8 3 1 7	SI	$5 + 7 = 12$
6 11 10 5 2 3 5 7	NO	
4 22 5 5 5 11	NO	

Límites

- $1 \leq N, K \leq 200$
- $1 \leq A_i \leq 200$

Subtareas

- (30 pts) $1 \leq N \leq 20$
- (30 pts)
  - $1 \leq N \leq 30$
  - $1 \leq K \leq 9$
- (40 pts) Sin restricciones adicionales

Solución

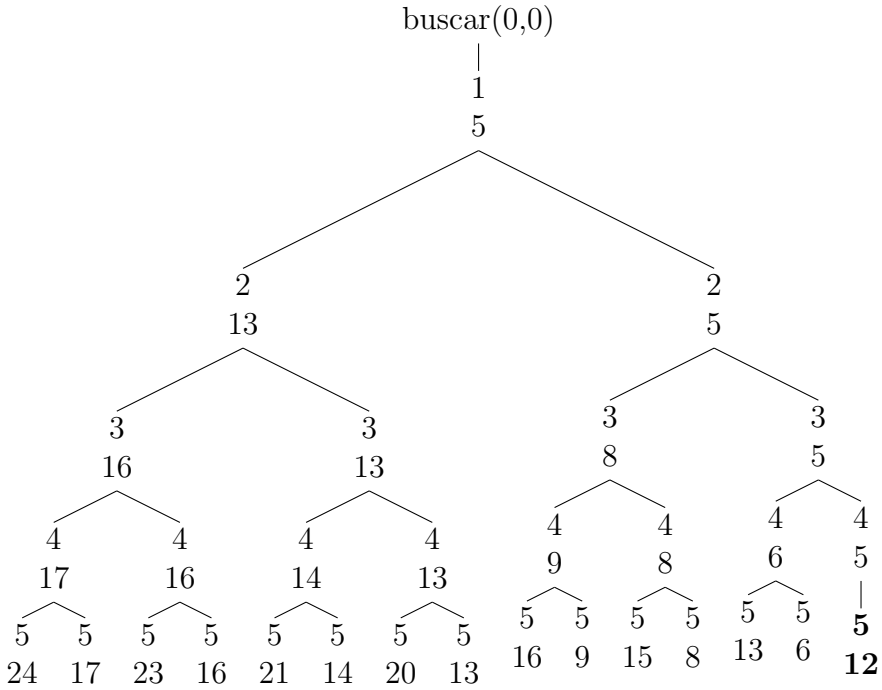
Vamos a ir construyendo una solución desde la fuerza bruta hasta obtener la de 100 puntos.

Fuerza bruta

La fuerza bruta es crear todos los subconjuntos posibles, viendo si la suma es lo que queremos o no, recordemos que en código se ve:

```
void buscar(int posicion, int suma) {
   if (posicion==N) {
      if (suma==K) {
         cout << "SI";
         exit(0);
      }
      return;
   }
   //prueba a incluir arr[pos]
   buscar(posicion+1, suma+arr[posicion]);

   //prueba a excluir arr[pos]
   buscar(posicion+1, suma);
}
```

Esta búsqueda nos produce el siguiente diagrama de recursión en el caso ejemplo uno. Cada llamada recursiva es representada por dos números, el primero es posición y el segundo es suma.



En total, se requirió de 31 llamadas recursivas y revisamos 15 subconjuntos, ¿pero realmente fue necesario hacer todo lo que hicimos?

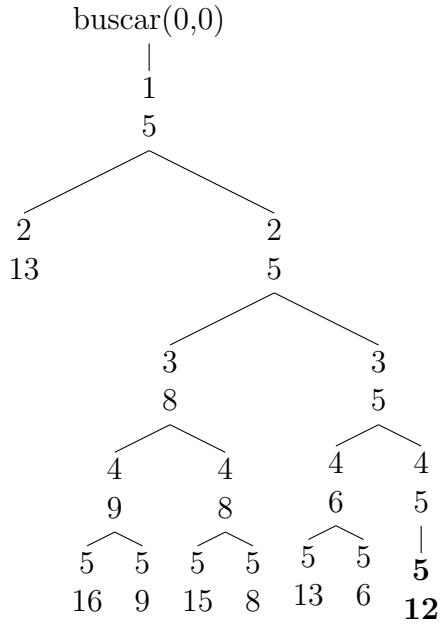
Primera poda

Pongamos atención en la llamada de `buscar(2, 13)`, desde aquí hacemos 14 llamadas recursivas, que si lo pensamos ya son innecesarias. Si la suma ya supero 12, que es lo que buscamos y no hay números negativos, entonces no importa que hagamos, jamas encontraremos 12.

Por lo tanto, la primera forma de mejorar la fuerza bruta es detener la búsqueda si superamos K. Esto se ve en código:

```
void buscar(int posicion, int suma) {
   if (suma > K) {
      return;
   }
   if (posicion==N) {
      if (suma==K) {
         cout << "SI";
         exit(0);
      }
      return;
   }
   //prueba a incluir arr[pos]
   buscar(posicion+1, suma+arr[posicion]);

   //prueba a excluir arr[pos]
   buscar(posicion+1, suma);
}
```

De forma que ahora el diagrama de la recursión es mucho menor.



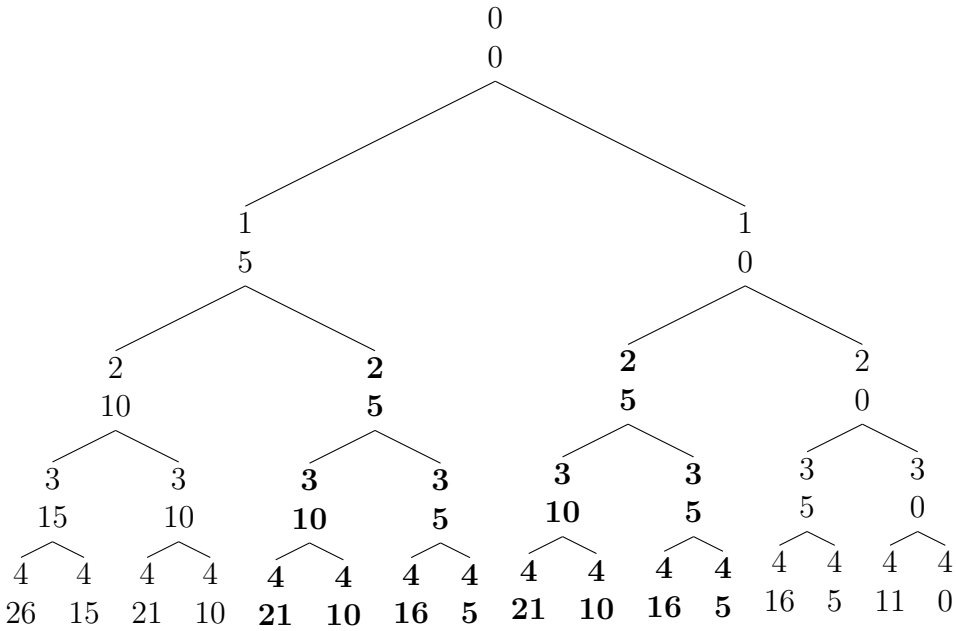
Esta poda nos permite sacar la segunda subtarea, esto es porque como K vale a lo más 9, la cantidad de subconjuntos por explorar son aquellos con 9 elementos o menos, los cuales para $n = 30$ hay $22,964,087$, esto lo podemos comprobar con una búsqueda que cuente la cantidad de subconjuntos de menos de 9 elementos, o con fórmulas matemáticas que veremos más adelante.

A esta forma de evitar visitar candidatos le llamamos poda, porque estamos podando ramas del diagrama y reduciendo el volumen de búsqueda. Igual que podar un árbol reduce la cantidad de hojas y el volumen de un árbol.

Segunda poda

Esta no es la única forma de podar, también podemos cortar la búsqueda en con información de lo que ya hemos explorado.

Para ver esto, veamos el tercer caso ejemplo.



Este diagrama no es afectado por la poda anterior, pero vamos a introducir una nueva poda. Para esto, probablemente te fijaste en que en el diagrama anterior, resaltamos dos ramas del diagrama en negritas.

El motivo por el cual están calcadas es porque son iguales, inician con buscar(2, 5) para llamar a buscar(3, 10) y buscar(3, 5), etc. Y como son iguales, entonces la segunda vez que lo exploremos, ¿podremos ahora encontrar la respuesta? Claro que no, vamos a ver lo que ya habíamos

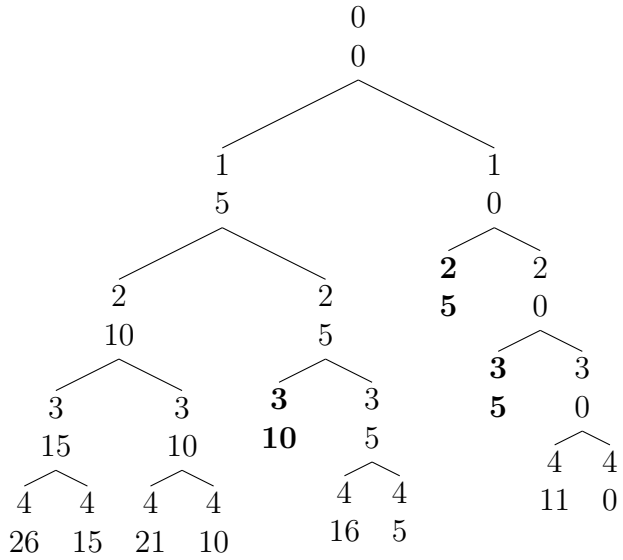
revisado antes.

Por esto, la segunda poda será que en caso de llegar a un buscar con un valor repetido de posicion y suma, detener la búsqueda.

En código se ve:

```
bool marcas[250][250];
void buscar(int posicion, int suma) {
   if (suma > K)
      return;
   if (marcas[posicion][suma])
      return;
   marcas[posicion][suma]=true;//Marcamos esta pareja de
       (pos, suma) como ya explorado para no repetir.
   if (posicion==N) {
      if (suma==K) {
         cout << "SI";
         exit(0);
      }
      return;
   }
   //prueba a incluir arr[pos]
   buscar(posicion+1, suma+arr[posicion]);

   //prueba a excluir arr[pos]
   buscar(posicion+1, suma);
}
```

Y esto produce el siguiente diagrama, donde marcamos con negritas las ramas podadas:



Esta poda nos pasa de 31 llamadas recursivas y 16 subconjuntos revisados, a solo usar 21 llamadas y revisar 8 subconjuntos.

Esta poda en concreto nos reduce la complejidad a $O(NK)$, esto es porque vamos a explorar solo cuando los argumentos aparecen por primera vez. Los valores de posición son de 0 a N y los la suma tiene rango de 0 a K. Lo cual implica que en total tenemos $(N+1)(K+1)$ posibles pares de valores.

Hay llamadas repetidas para los pares, pero estas no hacen nada ya que son podadas de forma inmediata, además, estas solo pueden ser llamadas por un buscar no podado.

Complejidad de $O(NK)$ corre en tiempo sin problemas para los 100 puntos.

Veamos otro ejemplo.

Ejemplo: Explorando una matriz

Un explorador tiene que decidir donde construir un camino para llegar de un punto A a un punto B.

Para auxiliarse el explorador ha hecho un mapa con los obstáculos que existen. Cuadriculó su mapa y quiere un camino que pase por el menor número de cuadros. El camino sólo puede ir de un cuadro a otro si tienen un lado en común, es decir, no puede avanzar en diagonal, y no puede pasar por un cuadro que contenga un obstáculo.

Cada cuadro del mapa se identifica por sus coordenadas, primero la fila y después la columna. Las filas están numerados de arriba hacia abajo iniciando con el 0. Las columnas están numeradas de izquierda a derecha iniciando con el 0.

Escribe un programa que dado un mapa con obstáculos encuentre el menor número de cuadros por los que debe pasar un camino que vaya del punto A al punto B, incluyendo a los cuadros que contienen a A y a B.

Entrada
En la primera línea los enteros N y M el número de filas y columnas del mapa. En cada una de las siguientes.

En las siguientes N líneas hay M enteros que pueden ser `0` ó `1`. Es `0` si no hay obstáculo en el cuadro correspondiente y `1` si lo hay.

En la siguiente línea (la penúltima) la fila y columna del punto A.

En la última línea la fila y columna del punto B.

Salida
En la primera linea el número de cuadros por los que pasa un camino

mínimo entre A y B

Ejemplo

Entrada	Salida
4 5 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 3 1 0 2	9

Límites

- $1 \leq N, M \leq 50$

Fuente: OMI 1996, Modificado subtask A

omegaup.com/arena/problem/OIEG2013SSC

Solución

Comencemos con la fuerza bruta, básicamente, buscaremos la secuencia de pasos más corta para llegar desde el punto A hasta el punto B.

Para esto, una observación clave es que ningún camino que sea el más corto pasa dos veces por la misma casilla, esto es porque es innecesario caminar en círculos.

Esto implica que la distancia a lo más usa todas las casillas, esto es $50 \times 50 = 2500$.

Ahora, la fuerza bruta la haremos con una función recursiva que vaya probando todas las secuencias de movimientos posibles, llevando cuenta de las coordenadas de donde va y cuantos pasos ha dado:

```
int N, M;
int respuesta=1e9;
int destino_i, destino_j;
int marcas[55][55];
void explorar(int pos_i, int pos_j, int pasos) {
   if (pos_i < 0 || pos_i >=N)
      return;//Estas fuera del mapa
   if (pos_j < 0 || pos_j >=M)
      return;//Estas fuera del mapa
   if (pos_i==destino_i && pos_j == destino_j) {
      //Comparar respuesta con la que encontramos
      if (respuesta > pasos)
         respuesta=pasos;
      return;
   }
   if (mapa[pos_i][pos_j])return;
   if (marcas[pos_i][pos_j]) return;
   marcas[pos_i][pos_j]=true; //Marcar para no pasar por aqui
       otra vez.
   explorar(pos_i-1, pos_j, pasos+1);//arriba
   explorar(pos_i+1, pos_j, pasos+1);//abajo
   explorar(pos_i, pos_j-1, pasos+1);//izquierda
   explorar(pos_i, pos_j+1, pasos+1);//derecha
   marcas[pos_i][pos_j]=false; //Desmarcar para permitir
       pasar por aqui en otros caminos.
}
int main() {
   cin >> N >>M;
   [...]
   int pi, pj;
   cin >> pi >> pj;
   cin >> destino_i>>destino_j;
   explorar(pi, pj, 1);
   cout << respuesta;
}
```

Esta fuerza bruta prueba todos los caminos posibles del punto A al punto B, pero es excesivamente lenta y obtenemos 0 puntos, necesitamos mejorarla.

La primera poda que podemos hacer es que si en algun momento llevamos más pasos que la respuesta encontrada hasta ahora, nos detengamos.

```
if (pasos >=respuesta) return;
```

Con esta idea mejoramos a 25 puntos, pero todavía podemos mejorar.

Para esto, veamos que si en algun momento llegamos a la casilla (i, j) con 20 pasos y luego llegamos con 50, de aquí no podremos encontrar un camino más rápido al punto B, pues todos los caminos desde aquí ya fueron explorados y de forma mejor.

Entonces, añadiremos para cada casilla un marcador de con que tiempo llegamos y solo exploraremos una casilla si llegamos a ella más rápido que antes.

```
if (tiempo[pos_i][pos_j]<=pasos) return;
tiempo[pos_i][pos_j]=pasos;
```

Esta segunda poda nos pasa la complejidad a: $O(distanciaMaxima \times N \times M)$ y como ya vimos, distanciaMaxima es $N \times M$, por lo tanto la complejidad es $O((NM)^2)$, lo cual equivale a $50^4 = 6,250,000$ que corre en tiempo.

Esta complejidad es resultado de que cada casilla es explorada a lo más la distancia máxima de veces.

2.1. Código general

Entonces, un código de backtracking se va a ver con la siguiente forma:

```
void buscar(solucion) {
   if (se poda solucion)
      return;
   if (solucion es valida) {
      registrar solucion;
   }
   buscar(soluciones siguientes)
}
```

Problemas de práctica

Problema 2.1 *Suma de digitos*
(omegaup.com/arena/problem/Suma-de-digitos)

Problema 2.2 *8 Reinas*
(omegaup.com/arena/problem/8Reinas)

Capítulo 3

Búsqueda en profundidad (DFS)

La búsqueda en profundidad, o DFS es un tipo de búsqueda basada en el Backtracking que vimos en el capítulo anterior, lo que la caracteriza es no explorar el mismo lugar dos veces.

El código de una DFS en general se verá de la siguiente forma:

```
void DFS(estado) {
   if (estado fue visitado)
      return;
   marca el estado como visitado;
   for (transiciones del estado) {
      DFS(transicion);
   }
}
```

Veamos un problema ejemplo:

Ejemplo: Dos operaciones

Javier tiene una calculadora un poco peculiar. Esta tiene un entero x en la pantalla y dos botones.

- El primer botón, al ser presionado le suma a al número x.
- El segundo botón, al ser presionado le suma $\frac{x}{b}$ a x, este botón solo puede ser usado cuando x es múltiplo de b.

Conociendo a y b, determina si de el valor inicial de x, se puede llegar a tener el número y en la calculadora.

Además, si se puede imprime como.

Entrada
Recibes cuatro enteros x, y, a y b. El valor inicial de la calculadora, el valor deseado, el valor de a, y de b; respectivamente.

Salida
Si es imposible convertir x en y, deberás imprimir `NO`.

Si es posible, deberás imprimir un entero K representando en cuantos pulsaciones de botón lo puedes hacer.

En la siguiente línea imprimirás K enteros, siendo las pulsaciones a los botones que debes hacer en orden de izquierda a derecha. Un `1` es presionar el primer botón, y un `2` representa el segundo botón.

Si hay varias respuestas, imprime cualquiera. (Ojo: no necesitas minimizar K).

Ejemplo

Entrada	Salida	Expliación
1 20 4 5	6 1 2 1 2 1 1	Presiona el primer botón, ahora tienes 5. Usa el segundo, ahora vale 6. Pulsa el primer botón, obtienes 10. Utiliza el segundo para tener 12. Usa el primer botón, obtén 16. Termina con el primero, llegamos a 20.
1 32 3 2	NO	Es imposible obtener 32.

Límites

- $1 \leq x, y, a, b \leq 10^5$

TODO ENLACE

Solución

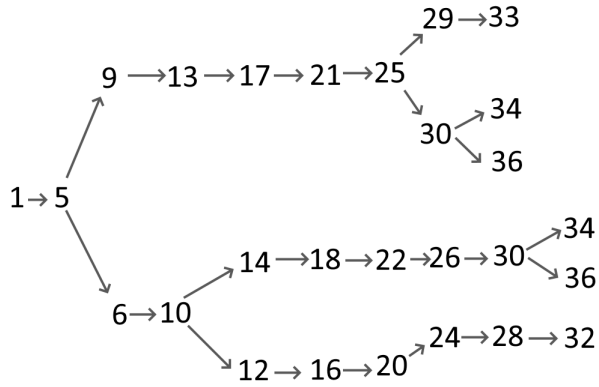
Iniciemos con la fuerza bruta.

Podemos ver que cada paso tenemos que tomar dos decisiones, o usamos el botón 1 o el 2. Podemos hacer una búsqueda exhaustiva para ver cual decisión tomar.

```
void exhaustiva(int x, int pasos) {
   if (x==y) {
      cout << pasos<<"\n";
      for (int i =0; i <pasos; i++) {
         cout<<solucion[i]<<" ";
      }
      exit(0);
   }
   if (x>y) {
      //x solo crece, de aqui ya es imposible hallar a y.
      return;
   }
   //presiona el boton 1.
   solucion[paso]=1;
   exhaustiva(x+a; pasos+1);,

   //presiona el boton 2 si es posible.
   if (x%b==0) {
      solucion[paso]=2;
      exhaustiva(x+x/b, pasos+1);
   }
}

int main() {
   [...]
   exhaustiva(0,0);
   cout << "NO";
}
```

Sin embargo, como ya hemos visto, la complejidad de la búsqueda exhaustiva es exponencial, por lo que no correrá para los límites de 10^5 que pide este problema.

Pero, ahora dibujemos lo que hace la búsqueda exhaustiva en el primer caso para ver si podemos mejorarlo.

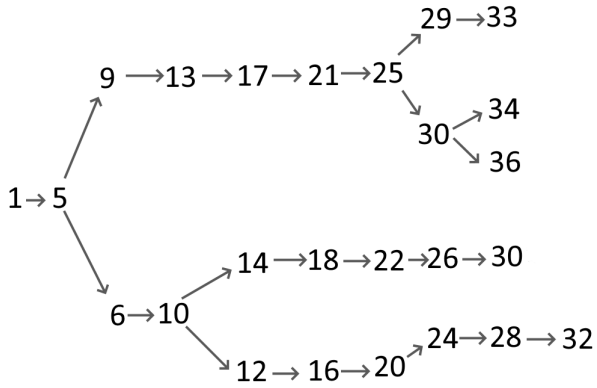


Allí vemos que podemos llegar al 30 con dos rutas. Pero sin importar cual ruta sigamos, de allí solo podemos llegar a los mismos números, al 34 y al 36.

Por lo tanto pregunto ¿realmente tiene sentido la segunda vez revisar el 30?

No, no lo tiene. Al llegar por segunda vez podemos podar la búsqueda, pues ya hemos visto todas las opciones del 30 antes, lo cuál nos ahorrará operaciones.

Ahora nuestra recursión se vería:



Puede que esto no se vea impresionante en el primer caso ejemplo, pero en el segundo caso ejemplo sí evitamos explorar mucho. Con esta regla, en el segundo ejemplo pasamos de 88 llamadas a la recursión a solo 31.

Entonces, para aplicar la nueva regla podemos tener un arreglo de booleanos llamado `visitado` que lleve cuenta de cuales valores ya visitamos con la búsqueda.

Cuando visitemos un valor de x revisamos si fue visitado antes, si lo fue detenemos la recursión, si no marcamos a x como visitado y continuamos con la búsqueda.

Esto en código se ve como:

```
bool visitado[100005];
void busqueda(int x, int pasos) {
   if (x==y) {
      cout << pasos<<"\n";
      for (int i =0; i <pasos; i++) {
         cout<<solucion[i]<<" ";
      }
      exit(0);
   }
   if (x>y) {
      //x solo crece, de aqui ya es imposible hayar a y.
      return;
   }
   if (visitado[x])
      return;
   visitado[x]=true;
   //presiona el boton 1.
   solucion[paso]=1;
   busqueda(x+a; pasos+1);,

   //presiona el boton 2 si es posible.
   if (x%b==0) {
      solucion[paso]=2;
      busqueda(x+x/b, pasos+1);
   }
}

int main() {
   [...]
   busqueda(0,0);
   cout << "NO";
}
```

Y tal cual, esto es una DFS, una búsqueda que va probando todas las opciones de donde se encuentra actualmente, pero nunca repitiendo la búsqueda en lugares ya visitados.

Analicemos la complejidad de este algoritmo, aquí es donde esta lo genial de este tema.

Veamos que este código solo ejecuta la parte de probar al botón 1 y 2 una sola vez por cada valor de x posible.

Y la parte de arriba solo es ejecutada tantas veces como búsqueda sea llamada(), que ya vimos que esta limitada a los posibles valores de x.

Como nuestros valores de x varían desde 1 hasta y, la complejidad de este algoritmo es $O(y)$. Lo cual significa que pasamos de un algoritmo exponencial a uno lineal. He aquí la hermosura de la búsqueda en profundidad, o DFS.

3.1. Estados y transiciones

Para entender completamente la DFS necesitamos definir un poco más sobre lo que explora.

En vez de llamar al valor de x que llevamos en la búsqueda como "lugar", o cualquier otra cosa, ahora le llamaremos por su nombre más formal de estado.

Y las "opciones"que tenemos en cada decisión le llamaremos por su nombre correcto, transiciones.

De forma que la DFS explora un espacio que esta conformado por estados conectados entre ellos por transiciones. Cada estado tendrá sus transiciones que nos permitirán alcanzar a otros estados.

En el código recursivo que hemos hecho de DFS, los estados vienen en los argumentos de la recursión y las transiciones son las llamadas recursivas que hacemos.

Entonces, nuestra búsqueda DFS explora todos los estados alcanzables desde uno inicial, utilizando las transiciones disponibles para pasar de estado a estado.

3.2. Complejidad

La complejidad de una DFS es la cantidad de estados más transiciones, llamemos al número de estados V y E al número de transiciones total. La DFS corre en $O(V + E)$.

En el ejemplo anterior teníamos y estados, así como a lo más $2y$ transiciones (dos por cada estado),por lo tanto la complejidad del ejemplo es: $O(V + E) = O(y + 2y) = O(y)$.

Problemas de práctica

Problema 3.1 *Subiendo la torre II*
(TODO)

Problema 3.2 *Camino en la matriz*
(TODO)

Problema 3.3 *Lugares alcanzables*
(TODO)

Problema 3.4 *Contando colonias*
(TODO)

Capítulo 4

Búsqueda en anchura (BFS)

La BFS es similar a la DFS, en el sentido que explora todos los estados a los que se puede llegar a partir uno inicial.

Pero la diferencia es que esto los explora un orden diferente.

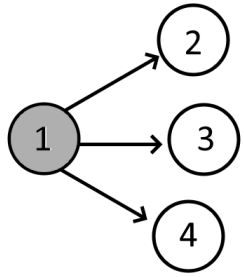
La DFS lo que hace es en un estado, explora totalmente lo que ofrezca una transición lo más profundo que puede hasta ya no poder explorar más y luego da vuelta atrás para ver transiciones que dejo pendiente.

En cambio, la BFS lo que hace es que hace es primero explorar el estado inicial, luego todos los estados que fueron descubiertos desde el inicial, luego los que fueron descubiertos en el paso anterior y así seguir.

Mostremos un diagrama del comportamiento de la BFS, en el diagrama los estados son representados por círculos. Si han sido explorados los marcamos de gris, pero si están en la lista por explorar, los deja-

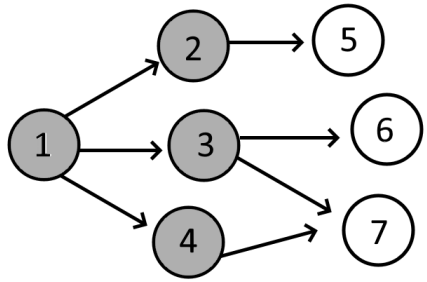
mos blancos. Las flechas representan las transiciones. Finalmente, los estados están enumerados en el orden que fueron alcanzados por la BFS.

Iniciamos explorando el estado inicial, 1 y poniendo los estados que alcanza en la lista por explorar.



Por explorar: 2 3 4

Y luego para cada una de esos estados los exploramos también y los nuevos que encontremos los vamos poniendo en la lista de .estados por explorar".



Por explorar: 5 6 7

De esta forma, comenzamos visitando todos los lugares alcanzables con una transición, luego dos transiciones, después tres, cuatro, cinco y así sucesivamente.

Y precisamente, este orden en el que exploramos nos da una buena ventaja. A todo estado llegamos con la menor cantidad de transiciones posibles. Lo cual podemos guardar ya que esto llega a ser información útil para muchos problemas.

Entonces, el pseudocódigo de una BFS se ve de la siguiente forma

```
bool visitado[];
int no_transiciones[];
explorando[];
porExplorar[];

void BFS(inicio) {
   explorando[0]=inicio;
   visitado[inicio]=true;
   no_transiciones[inicio]=0;
   int numExp=1;//No de elemntos en la lista explorando
   int numPorExp=0;//No de elemntos en la lista porExplorar
   //mientras explorando no este vacio
   while (numExp>0) {
      //Encontar los estados alcanzables desde explorando
      for (int i =0; i < numExp; i++) {
         estado= explorando[i];
         for (transicion del estado) {
            if (visitado[transicion]==false) {
               no_transiciones[transicion]=
                  1+no_transiciones[estado];
               visitado[transicion]=true;
               porExplorar[numPorExp]=transicion;
               numPorExp++;
            }
         }
      }
      //Pasar porExplorar a explorando para la siguiente
          iteracion
      for (int i=0; i < numPorExp; i++)
         explorando[i]=porExplorar[i];
      numExp=numPorExp;
      numPorExp=0;
   }
}
```

Como ya es de costumbre, veamos un ejemplo para entender esto.

Ejemplo: Dos operaciones

Javier tiene una calculadora un poco peculiar. Esta muestra un número x en pantalla y tiene dos botones.

- El primer botón le suma a al valor de x.
- El segundo botón le suma $\frac{x}{b}$ a x, pero solo puede ser presionado cuando x sea un múltiplo de b.

Ahora Javier se pregunta cuantas veces debe presionar un botón para que el valor x se convierta en y.

Entrada
La entrada constará de cuatro enteros: x, y, a y b. El valor inicial de la calculadora, el valor deseado, y el valor de a y b para los botones.

Salida
Imprime un entero que sea la cantidad de pulsaciones mínima para convertir x a y. Si es imposible pasar de x a y imprime -1.

Ejemplo

Entrada	Salida	Expliación
1 20 4 5	6	Presiona el primer botón, ahora tienes 5. Usa el segundo, ahora vale 6. Pulsa el primer botón, obtienes 10. Utiliza el segundo para tener 12. Usa el primer botón, obtén 16. Termina con el primero, llegamos a 20.
1 32 3 2	-1	Es imposible obtener 32.

Límites

- $1 \leq x, y, a, b \leq 10^5$

ENLACE: TODO

Solución

Entonces, en este problema nos piden la mínima cantidad de operaciones para pasar de x al valor de y.

Recordemos que la BFS es perfecta para esta situaciones, pues calcula la mínima cantidad de transiciones para pasar de un estado inicial a todos los demás, incluyendo y.

Para esto haremos una BFS que use de estados el número de la calculadora y de transiciones los botones. Esta explorará todas las operaciones desde x hasta llegar a y.

Esto se verá:

```
bool visitado[100005];
int distancia[100005];
int explorando[100005], porExplorar[100005];
int bfs(int x, int y, int a, int b) {
   explorando[0]=x;
   visitado[x]=true;
   distancia[x]=0;
   int numExp=1, numPorExp=0;
   while (numExp>0) {
      for (int i=0;i<porExp; i++) {
         int actual=explorando[i];
         if (actual==y)
            return distancia[x];

         int siguiente= actual+a;
         if (siguiente <= y && visitado[siguiente]==false) {
            visitado[siguiente]=true;
            distancia[siguiente]=1+distancia[actual];
            porExplorar[numPorExp++]=siguiente;
         }
         siguiente=actual+actual/b;
         if (siguiente <= y && actual%b ==0 &&
             visitado[siguiente]==false) {
            visitado[siguiente]=true;
            distancia[siguiente]=1+distancia[actual];
            porExplorar[numPorExp++]=siguiente;
         }
      }
      for (int i=0; i<numPorExp; i++)
         explorando[i]=porExplorar[i];
      numExp=numPorExp;
      numPorExp=0;
   }
   return -1;
}
```

4.1. Complejidad

La complejidad de una BFS es sencillamente de calcular. Veamos que cada estado es colocado una sola vez dentro de la cola y por lo tanto, explorado una sola vez.

Por lo tanto, cada transición también es visitada una única vez, cuando su estado correspondiente sea visitado.

Esto provoca que la complejidad de la BFS sea $O(V + E)$, donde V es la cantidad de estados y E es el número de transiciones.

En el ejemplo 5.1, la cantidad de estados son a lo más y y a lo más tenemos dos transiciones por estado, por lo que la complejidad es $O(V + E) = O(y + 2y) = O(y)$.

Ejemplo: OMI-98 Explorador II

Un explorador tiene que decidir donde construir un camino para llegar de un punto A a un punto B.

Para auxiliarse el explorador ha hecho un mapa con los obstáculos que existen. Cuadriculó su mapa y quiere un camino que pase por el menor número de cuadros. El camino sólo puede ir de un cuadro a otro si tienen un lado en común, es decir, no puede avanzar en diagonal, y no puede pasar por un cuadro que contenga un obstáculo.

Cada cuadro del mapa se identifica por sus coordenadas, primero la fila y después la columna. Las filas están numerados de arriba hacia abajo iniciando con el 0. Las columnas están numeradas de izquierda a derecha iniciando con el 0.

Escribe un programa que dado un mapa con obstáculos encuentre el menor número de cuadros por los que debe pasar un camino que vaya del punto A al punto B, incluyendo a los cuadros que contienen a A y a B.

Entrada
En la primera línea los enteros N y M el número de filas y columnas del mapa. En cada una de las siguientes.

En las siguientes N líneas hay M enteros que pueden ser `0` ó `1`. Es `0` si no hay obstáculo en el cuadro correspondiente y `1` si lo hay.

En la siguiente línea (la penúltima) la fila y columna del punto A.

En la última línea la fila y columna del punto B.

Salida
En la primera linea el número de cuadros por los que pasa un camino mínimo entre A y B

Ejemplo

Entrada	**Salida**
4 5 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 3 1 0 2	9

Límites

- $1 \leq N, M \leq 750$

ENLACE TODO

Solución

Anteriormente vimos una solución con Podas para este problema con complejidad $O((NM)^2)$, pero esto no es suficiente para este problema.

Claramente, podemos hacer una BFS donde los estados sea cada punto de la cuadricula y que cuente cuantos pasos necesito para llegar a B iniciando desde A.

Entonces nuestros estados serán descritos con dos valores f y c, la fila y columna. Pero hasta ahora la BFS que vimos usa solo un número como valor, piensa en como adaptarla para que se puedan usar estados que usen dos valores y trata de resolverlo antes.

Usando dos valores

La forma de adaptar la BFS que vimos para funcionar con esto es bastante sencillo. Lo único que necesitamos hacer es que en vez de introducir un valor a la lista porExplorar, agregamos dos por cada estado.

De forma que si en porExplorar tenemos tres casillas, esta lista se vería {f1, c1, f2, c2, f3, c3}

Entonces, lo que podemos hacer es realizar la bfs. El código quedará aproximadamente:

```
bool visitado[755][755];
int distancia[755][755];
int BFS(int Afila, int Acolumna) {
   explorar[0]=Afila; explorar[1]=Acolumna;
   visitado[Afila][Acolumna]=true;
   int numExp=2; int numPorExp=0;
   int transicionesI[4] ={1,-1,0,0};
   int transicionesJ[4]={0,0,1,-1};
   for (int c =0; c < numExp; c+=2) {
      int i = explorando[c];
      int j = explorando[c+1];
      if (i==Bfila && j==Bcolumna) return distancia[i][j];
      for (int tr =0; tr < 4; tr++) {
         int ni= i+transicionesI[tr];
         int nj= j+transicionesJ[tr];
         if (0<=ni && ni < N && 0 <=nj && nj<M) {
            if (mapa[ni][nj]==0 && !visitado[ni][nj]) {
               visitado[ni][nj]=true;
               distancia[ni][nj]=1+distancia[i][j];
               porExplorar[numPorExp]=ni; //Agrega la
                   coordenada ni
               porExplorar[numPorExp+1]=nj; //Agrega la
                   coordenada nj
               numPorExp+=2;
            }
         }
      }
      [...]
   }
}
```

Problemas de práctica

Problema 4.1 *Two Buttons*
(codeforces.com/problemset/problem/520/B)

Traduccion: TODO

Problema 4.2 *Camino mas corto en matriz*
(TODO)

Problema 4.3 *Volcan*
(omegaup.com/arena/problem/Volcan)

Problema 4.4 *IOI 1994 - Relojes*
(omegaup.com/arena/problem/relojes)

Capítulo 5

Búsqueda binaria

Esta técnica es una de las más poderosas e importantes en la informática. Es una forma muy eficiente de hacer búsquedas que nos permite resolver problemas antes imposibles.

Utilizada en muchas aplicaciones, nos permite resolver problemas donde una búsqueda exhaustiva tardaría miles de millones de años en menos de un segundo. Y esta impresionante herramienta ahora será tuya.

La búsqueda binaria difiere de la fuerza bruta al evitar revisar absolutamente todas las opciones del espacio de búsqueda. Si no, lo que hace es que en cada paso logra descartar la mitad del espacio de búsqueda al darse cuenta que la respuesta no está allí.

Porque en cada paso va descartando la mitad de las opciones, llega a una única opción (la respuesta si esta existe) en muy poco tiempo. Mientras que la búsqueda ehaustiva va eliminando opciones de una en

una, lo cual es lento.

Comparemos un ejemplo de la búsqueda exhaustiva vs la binaria, si tuviéramos 128 candidatos.

Número de pasos	Candidatos restantes	
	Exhaustiva	Binaria
0	128	128
1	127	64
2	126	32
3	125	16
4	124	8
5	123	4
6	122	2
7	121	1
8	120	-
	• • •	
125	3	-
126	2	-
127	1	-

Como vemos, búsqueda binaria pudo reducir los candidatos a solo uno con siete pasos, mientras que la búsqueda completa requirió de 127.

El motivo por el cuál la búsqueda binaria es tan efectiva, es que realiza $\lceil log_2(candidatos) \rceil$ pasos [1]. Esto es porque $\lceil log_2 \rceil$ nos permite calcular cuantas veces podemos dividir un número entre dos hasta que sea 1.

Veamos un ejemplo que se puede resolver con búsqueda binaria.

[1] $\lceil log_2(A) \rceil$ significa: techo del logaritmo base dos de A.
Recordemos que $log_2(A) = x$ significa que $2^x = A$.
Y el techo nos dice que tomemos el menor entero que sea mayor igual que el valor de adentro, por ejemplo:. $\lceil 3.12 \rceil = 4$

Ejemplo: Del área al perímetro

Javier es un granjero y esta cansado de que sus animales siempre huyan de su granja, por lo que ha decidido poner una reja al rededor de todo el terreno.

Sin embargo, Javier no sabe cuantos metros de reja va a necesitar y te ha contratado para que tu le digas esto.

Lo que él si sabe es que el tiene forma cuadrada con lados de longitud entera, además recuerda que este mide A metros cuadrados de área.

Ayuda a Javier para que sepa cuanta reja necesita.

avier es un granjero y esta cansado de que sus animales siempre huyan de su granja, por lo que ha decidido poner una reja al rededor de todo el terreno.

Sin embargo, Javier no sabe cuantos metros de reja va a necesitar y te ha contratado para que tu le digas esto.

Lo que él si sabe es que el tiene forma cuadrada con lados de longitud entera, además recuerda que este mide A metros cuadrados de área.

Ayuda a Javier para que sepa cuanta reja necesita.

Entrada
Un entero A, el tamaño del terreno en metros cuadrados.

Salida
Un entero indicando cuantos metros de reja necesita para rodear todo el terreno.

Ejemplos

Entrada	Salida
36	24
100	40
1	4

Límites

- $1 \leq A \leq 10^{18}$

Subtareas

- (35 pts) $1 \leq A \leq 10^9$
- (65 pts) Sin restricciones adicionales

ENLACE: TODO

Solución

Como siempre, antes de leer la solución te invitamos a que intentes el problema.

En este problema nos piden de que dado el área de un cuadrado, imprimamos el perímetro. Por esto, recordemos las fórmulas del cuadrado.

$$Perímetro = 4 \times Lado$$

$$Área = Lado \times Lado$$

Entonces, si encontramos el lado que nos de el área de entrada, podremos encontrar el perímetro.

Para este problema vamos a ver dos estrategias para resolverlo, la de búsqueda lineal y la binaria.

Comencemos con la que ya deberíamos estar familiarizada, usemos búsqueda lineal.

Búsqueda lineal

Entonces, buscaremos el entero L de entre todos los posibles, que cumpla que $A = 4 \times L$. Para esto, podremos ir probando del 1 en adelante hasta encontrar el que cumpla.

```
long long L=1;
while (L*L != A) {
   L++;
}
```

Esto itera por todos los enteros del 1 a la respuesta que es $\sqrt{A}$. Por lo tanto, su complejidad es $O(\sqrt{A})$. Lo cual es suficiente para la subtarea de 35 puntos, pero no para el límite completo de 10^{18}.

Búsqueda binaria

Ahora resolvamos el problema utilizando búsqueda binaria.

Definamos nuestro espacio de búsqueda, ¿cuáles valores puede ser L? y si lo pensamos puede ser desde 1 hasta 10^9. Porque $10^9 \times 10^9 = 10^{18}$ y de los límites sabemos que $1 \leq A \leq 10^{18}$.

Entonces queremos encontrar la respuesta que esta entre 1 y 10^9, hagamos una función que logre esto llamada buscar que reciba el rango de donde esta la respuesta.

```
//Encuentra L tal que L*L=A, sabiendo que a<=L<=b.
long long buscar(long long a, long long b, long long A);
```

Ahora tenemos 10^9 candidatos donde encontrar la respuesta y queremos reducirlo a la mitad.

Para esto podemos ver que sucede con 5×10^8.

Si resulta que $(5 \times 10^8) \times (5 \times 10^8) < A$, entonces sabemos que cualquier valor menor igual que 5×10^8 no funcionará porque es demasiado pequeño. Por lo tanto la respuesta debe estar entre $5 \times 10^8 + 1$ y 10^9 y los candidatos se redujeron a la mitad.

Pero si en vez sucede que $(5 \times 10^8) \times (5 \times 10^8) >= A$, entonces sabremos que cualquier valor más grande que 5×10^8 nos dará valores más grandes que A y por lo tanto la respuesta no estará allí. Ahora nuestros candidatos son los números entre 1 y 5×10^8, reduciendo los valores que podrían ser la respuesta a la mitad.

Y de hecho, si en general, la respuesta esta entre a y b, nos convendrá preguntar por el punto medio $(a + b)/2$, que nos reducirá el espacio a la mitad.

Una vez que redujimos el rango de la búsqueda, tendremos que encontrar la respuesta en ese nuevo rango, y para esto podemos hacer recursión.

De forma que ahora tenemos:

```
long long buscar(long long a, long long b, long long A) {
   long long m =(a+b)/2;
   if (m*m < A) {
      return buscar(m+1, b, A);
   } else {
      return buscar(a, m, A);
   }
}
```

Ahora, lo que le falta a esa recursión es una condición de paro, saber cuando ya termino y encontramos la respuesta.

Podremos ver que habremos encontrado la respuesta cuando ya estemos seguros de cual es esta. Y esto sucede cuando nuestro rango solo incluye un valor, es decir, cuando $a == b$ se cumpla.

```
long long buscar(long lonng a, long long b, long long A) {
   if (a==b)
      return a;
   long long m=(a+b)/2;
   if (m*m<A) {
      return buscar(m+1, b, A);
   } else {
      return buscar(a, m, A);
   }
}
```

Y con esto, tenemos que `L=buscar(1, 1000000000, A)`.

Ahora, la complejidad de esto es $O(log(\sqrt{A}))$, lo cual corre perfectamente para 10^{18}.

Nota: En este problemas también se pudo haber usado `L=sqrt(A)`

con la librería <math.h> que calcula el valor rápidamente, pero se uso búsqueda binaria para ejemplificar.

También puede ser escrita de forma iterativa de la siguiente manera:

```
long long buscar(long lonng a, long long b, long long A) {
   while (a!=b) {
      long long m=(a+b)/2;
      if (m*m < A) {
         a=m+1;
      } else {
         b=m;
      }
   }
   return a;
}
```

Ejemplo: Buscar en un arreglo

Se te da un arreglo A de N enteros diferentes. El arreglo estará en orden creciente, es decir, $A[i] < A[i+1]$.

Deberás responder T preguntas:

Cada pregunta consistirá de un entero q_i y tu deberás imprimir el índice del valor q_i o -1 si este valor no existe. **Entrada**
El enteros N.

En la siguiente línea: N enteros separados, los valores del arreglo A.

En la siguiente línea recibirás el entero T.

En las siguientes T líneas recibirás los valores de cada pregunta.

Salida
Imprime la respuesta cada pregunta en una línea en el mismo orden que el de lectura.

Ejemplo

Entrada	Salida
7 2 5 6 7 8 9 10 5 2 6 4 10 5	0 2 -1 6 1

Límites

- $1 \leq N, T \leq 10^5$
- $1 \leq A[i], q_i \leq 10^9$

Enlace: TODO

Solución

Esta ocasión nos piden encontrar el indice de un valor en un arreglo ordenado. Ya hemos visto como hacerlo con búsqueda lineal:

```
int indice(int q) {
   for (int i =0; i < N;i++)
   if (A[i]==q)
   return i;
   return -1;
}
```

Sin embargo, esto no corre en tiempo ya que la complejidad es $O(N)$ por pregunta, siendo en total $O(TN)$ y como $TN = 10^{10}$, obtendremos TLE.

Pero veamos que podemos usar búsqueda binaria. ya que al preguntar por una posición de en medio y discernir si tenemos que buscar adelante o atrás.

```
int binaria(int a, int b, int q) {
   if (a>b)
   return -1;
   int m =(a+b)/2;
   if (A[m]==q)
   return m;
   if (A[m]<q) {
      return binaria(m+1, b, q);
   } else {
      return binaria(a, m-1, q);
   }
}
int indice(int q) {
   return binaria(0, N-1, q);
}
```

La nueva complejidad ahora es $O(logN)$ por pregunta, en total $O(TlogN)$.

5.1. Dificultades

Ya vimos que la búsqueda binaria tiene una ventaja enorme sobre la búsqueda lineal ya que resuelve el problema en un tiempo mucho menor.

Pero tristemente, no siempre es posible aplicar la búsqueda binaria. Hay veces en las que no podemos descartar fácilmente la mitad de los candidatos.

Un caso donde puede suceder sería en el ejemplo anterior si el arreglo no estuviese ordenado. En ese caso no podríamos hacer el truco de solo revisar adelante si $A[m] < q$ ya que no nos da suficiente información esa pregunta para eliminar todos los valores de 0 a m.

Por supuesto, el problema anterior tiene corrección para que la búsqueda binaria siga funcionando y muchas veces esto es parte del problema, ¿cómo hago la binaria aquí? Pero hay otras veces que es imposible, o al menos, más allá de los conocimientos actuales.

En esos casos, no quedará de otra más que hacer búsqueda completa o usar una técnica diferente a búsqueda.

Otro ejemplo donde la búsqueda binaria no se puede utilizar es en encontrar el primer divisor que no sea 1 de un entero. Es difícil descartar la mitad de candidatos en una sola operación.

5.2. Función de validación

Igual que en búsqueda lineal podíamos agregar funciones más complicadas a la hora de buscar la respuesta, lo mismo sucede en búsqueda binaria. A veces requeriremos de más código que una simple comparación para saber en cual mitad esta la respuesta.

Veamos unos ejemplos.

Ejemplo: Bicicleta de Karel II

Karel ha comprado una bicicleta eléctrica con la que planea completar un recorrido. El recorrido se puede ver como N colinas en línea recta tal que la i-ésima colina tiene altura h_i. Karel comienza en la colina hasta la izquierda y quiere terminar en la ultima colina de hasta la derecha.

Cuando Karel sube un metro gasta 1 unidad de energía, mientras que bajar un metro recupera 1 unidad de altura. Si Karel en algún momento necesita subir, pero su batería tiene 0 de energía, Karel se quedará atorado y no terminará el recorrido.

Por suerte al inicio hay una estación de recarga donde Karel puede recargar su bicicleta. Como nota, la batería tiene capacidad R y jamás podrá almacena más energía que R.

Actualmente Karel tiene 0 de energía, Determina cuál es la menor cantidad de energía que es necesaria recargar al inicio para completar el recorrido. O determina si es imposible hacer el recorrido con la bicicleta de Karel.

Entrada

La primera línea tiene dos enteros, el valor de N y R.

En la siguiente línea vienen N, enteros separados por espacios, siendo la altura de las colinas de izquierda a derecha. Recuerda que Karel comienza en la primera colina y quiera terminar en la última.

Salida

Un entero, representando la menor cantidad de energía necesaria para completar el recorrido. Si Karel no puede completar el recorrido, imprime -1.

Casos ejemplo

Entrada	Salida	Expliación
6 8 4 6 3 5 7	3	Karel inicia con 3 de energía, moverse de la primera a la segunda colina le toma 2, ahora tiene 1. Luego avanza y se recarga 3, ahora tiene 4. Después continua y se consume 2, ahora tiene 2. Vuelve a avanzar quedándose con 0 de energía. Pero luego avanza y se recarga a 5. Finalmente avanza para termina con 5.
5 6 1 10 1 2 0	-1	

Límites

- $2 \leq N, R \leq 10^5$

- $0 \leq h_i \leq 10^9$

Fuente: OMIS online 2022.
(omegaup.com/arena/problem/Bicicleta-de-Karel)

Solución

Este problema 1.6 de búsqueda lineal con validación en la página **??**, si no sabes resolverlo con búsqueda lineal para los límites de allí, primero descubre esa solución.

Esta vez, los límites son más estrictos, de forma que la solución estándar con búsqueda lineal no funciona, pero veamos cual es porque nos será útil.

La respuesta siempre estará entre 0 yR. La búsqueda lineal funciona de la siguiente manera.

```
int respuesta=-1;
for (int e=0; e<=R; e++) {
   if (funciona(e)) {
      respuesta=e;
      break;
   }
}
```

Y la función `bool funciona(int e)` te regresa verdadero si Karel puede completar el recorrido comenzando con e de energía.

Esta función simplemente simula el recorrido para ver si Karel se atora en algún momento. Se ve de la siguiente forma:

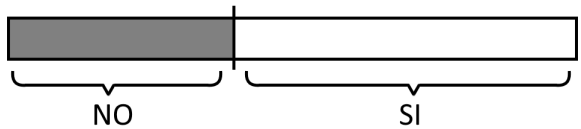
```
bool funciona(int e) {
   for (int i = 1; i <N; i++){
      e-=A[i]-A[i-1];
      if (e > R)
      e=R; //Limita la energia
      if (e < 0)
      return false;
   }
   return true;
}
```

La función `funciona()` tiene una complejidad de $O(N)$ y como es llamada en R valores, la complejidad total es $O(RN)$.

Pero ahora veamos dos hechos importantes:

- Si funciona(m) cumple, también lo hará cualquier valor mayor que m.
- Si funciona(m) falla, también lo hará cualquier valor menor que m.

Es decir, el rango de búsqueda se ve de la siguiente forma:



Esto hace que podamos hacer búsqueda binaria para encontrar el primer SI.

El código se verá como:

```
int a=0,b=R;
while(a!=b) {
   int m=(a+b)/2;
   if (funciona(m)) {
      b=m;
   } else {
      a=m+1;
   }
}
int respuesta=-1;
if (funciona(a))
   respuesta=a;
```

5.3. Complejidad

La complejidad de la búsqueda binaria es entonces $O(logN)$, donde N es el tamaño del espacio de búsqueda porque $logN$ nos dice cuantas veces N puede ser dividido entre dos.

Pero también, en cada paso de la búsqueda podemos usar una llamada de validación. En el ejemplo anterior estamos llamando a `funciona()` que tiene complejidad $O(N)$, por lo tanto, la complejidad total es $O(NlogR)$.

Es decir, la complejidad de una búsqueda binaria es $O(busqueda \times validacion)$ que es igual a $O(logN \times validacion)$.

5.4. Búsqueda en los reales

Hay veces en la que no estamos buscando un valor entero, si no en vez querremos encontrar un valor real a cierta precisión.

El problema dirá algo por el estilo de: imprime la respuesta con precisión de 10^{-6}, esto quiere decir que tu respuesta no debe estar alejada de la solución por más de 10^{-6}.

Para estos problemas debes cambiar la condición de paro de $a == b$, por $b - a < precision$. Aunque en realidad, recomiendo poner una precisión un poco más ajustada ya que no tiende a aumentar mucho el costo y permite una programación más relajada.

De forma que ahora la binaria se verá como:

```
double a =0, b=1e9;
double epsilon = 1e-8;
while(b-a>= epsilon) {
   double m = (a+b)/2;
   if (funciona(m)) {
      b=m;
   } else {
      a=m;
   }
}
```

Ejemplo: Raíz cuadrada real

Calcula la raíz cuadrada de un número A con precisión absoluta de 10^{-4}.

Ejemplo

Entrada	Salida
10	3.1622
16	4.0000

Límites

- $1 \leq A \leq 10^9$

Solución

Usamos las mismas observaciones que usamos para el ejemplo "De área a perímetro", pero esta vez lo haremos con `double` y nos detenemos basados en una precisión.

```
double raiz(double A) {
   double a=0, b=A;
   while (b-a >= 1e-5) {
      double m=(a+b)/2;
      if (m*m < A) {
         a=m;
      } else {
         b=m;
      }
   }
   return a;
}
```

Problema 5.1 *Edita el código para resolver el siguiente problema: Un prisma de altura $x+1$, ancho $2x+5$ y largo $3x+1$ tiene volumen n. Dado n imprime x. ($5 \leq n \leq 10^{18}$).*

5.4.1. Complejidad

La complejidad de este procedimiento sigue siendo $O(log(Espacio))$, pero ahora debemos encontrar el tamaño del espacio de búsqueda.

Con un poco de matemáticas veremos que es $E = \frac{R-L}{precision}$, en el caso de ejemplo la precisión es pues: $\frac{A}{10^{-4}}$, aunque nosotros usamos una precisión de 10^{-5}, por lo que es $\frac{A}{10^{-5}}$.

Por lo tanto, la complejidad del ejemplo anterior es:

$$
\begin{aligned}
& O(logE) \\
&= O(log\frac{A}{10^{-5}}) \\
&= O(log(A) - log(10^{-5})) \\
&= O(log(A) + log(10^{5}))
\end{aligned}
$$

Entonces, poner una precisión de 10^{-5} solo nos aumenta un costo de $log(10^5) \approx 16.6$, o en otras palabras, 17 preguntas extra.

Problemas de práctica

Problema 5.2 *Transporte de cajas*
(omegaup.com/arena/problem/Transporte-de-Cajas)

Problema 5.3 *Hommer y sus rosquillas*
(TODO)

Fuente: Adaptación de ORIG 2015-2016

Problema 5.4 *ConneR and the A.R.C. Markland-N*
(codeforces.com/problemset/problem/1293/A)

Traducción: TODO

Problema 5.5 *Barcos turísticos*
(omegaup.com/arena/problem/Barcos-turisticos)

Problema 5.6 *Subcadena de suma K*
(TODO)

Problema 5.7 *La Combi de Cesar*
(omegaup.com/arena/problem/La-Combi-de-Cesar)

Problema 5.8 *Carreras de la Formula π*
(TODO)

Problema 5.9 *Planetas*
(omegaup.com/arena/problem/planetas)

Problema 5.10 *Carretera*
(omegaup.com/arena/problem/carretera)

Problema 5.11 *Determina en que punto dos rectas se intersecan.*

Parte III

Matemáticas olímpicas

Parte IV

STL y estructuras de datos

Parte V

Grafos

Parte VI

Técnicas de resolución de problemas

Parte VII

Problemas no estándar