

Búsquedas

Héctor Fernando Ricárdez Lara

Índice

Introducción	1
Búsqueda lineal	3
Ejemplo 1.1	4
Ejemplo 1.2	5
Problemas de práctica	7
Búsqueda lineal con función de validación	9
Ejemplo 1.3	9
Conclusión	12
Problemas de práctica	13
Búsqueda completa	17
Pares de elementos	19
Ejemplo 2.1.1	19
Complejidad	22
Problemas de práctica	23
Subconjuntos	29
Definición	29
Buscar subconjuntos (ejemplo 2.2.1)	30
Complejidad	34
Ejemplo 2.2.2	35
Problemas de práctica	38
Decisiones	41

Ejemplo 2.3.1	43
Ejemplo 2.3.2	47
Complejidad	51
Problemas de práctica	51
Búsqueda binaria	57
Ejemplo 3.1	59
Ejemplo 3.2	63
Dificultades	65
Problemas de práctica	66
Función de validación	72
Ejemplo 3.4	72
Problemas de práctica	75
Búsqueda en los reales	78
Ejemplo 3.5	78
Problemas de práctica	80
Binaria en C++	83
Búsqueda en profundidad (DFS)	85
Ejemplo 4.1	86
Estados y transiciones	91
Complejidad	92
Problemas de práctica	93
Búsqueda en amplitud (BFS)	97
Ejemplo 5.1	99
Complejidad	103
Problemas de práctica	104
Problemas	111
Anexo	113
Problemas interactivos	113
Ejemplo	115
Compilar y probar interactivos	117

Colas	121
Usar queue	121

Introducción

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 este libro trabajaremos un poco más en desarrollar esta intuición e ideas.

Igual que en la vida diaria buscamos muchas cosas, en la resolución de problemas nos la pasamos buscando cosas, ya sea: la respuesta, algún valor en un arreglo, la raíz cuadrada de un número, etc.

Es decir, en estos problemas tendremos una lista de objetos y trataremos de encontrar en la lista uno que cumpla alguna propiedad específica. Es decir, queremos encontrar una respuesta dentro de una lista de candidatos.

Entonces, búsqueda es simplemente encontrar una respuesta dentro en una lista de posibles candidatos.

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 1.1

Descripción

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

Límites

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

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;  
}
```

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 1.2

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;
```

Problemas de práctica

Problema 1.1 Dado una lista de N enteros, cuenta cuantas veces aparece el valor K .

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

Enlace: TODO

Problema 1.2 Dado una lista de N enteros, imprime todos los múltiplos de 5 que estén en la lista y en el mismo orden que aparecen en la lista.

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

Enlace: [TODO]

Problema 1.3 Cuenta cuantos enteros positivos dividen a K exactamente. Similar al ejemplo, pero los límites son más grandes.

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

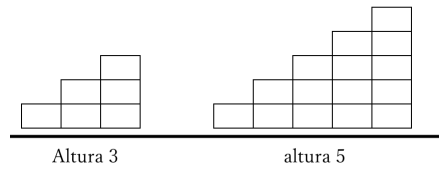
Enlace: 3 [TODO]

Problema 1.4 Encuentra el entero más grande que sea menor o igual a R y que la suma de sus dígitos sea menor o igual que S . Se garantiza que siempre existe este entero.

- $1 \leq S \leq 50$
- $1 \leq R \leq 10^5$

Enlace: omegaup.com/arena/problem/bl-1-4 [TODO]

Problema 1.5 Fernando construye escaleras de ladrillos de la siguiente forma:



El otro día, Fernando obtuvo K ladrillos, ahora se pregunta ¿Qué tan alto que puede construir una escalera? Dado K , responde su pregunta.

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

Casos ejemplo

Entrada	Salida
12	4
8	3
20	5

Enlace: [TODO]

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 1.3

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 $\min(A[1], A[2], \dots, A[N])$. 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 tamaño 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.

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: $O(\text{Busqueda} \times \text{Validar}) = O(AN)$.

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.

Conclusión

Habría veces que en la búsqueda tengamos que hacer código para validar cada contacto que encontremos, y la complejidad de la validación se meterá como un factor a la complejidad de la búsqueda.

Con esto obtenemos un algoritmo con complejidad $O(\text{Busqueda} \times \text{Validar})$.

Problemas de práctica

Problema 1.6 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 última 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 almacenar 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	Expiación
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

- $2 \leq N, R \leq 5000$
- $0 \leq h_i \leq 5000$

Fuente: OMIS online 2022.

Enlace: [TODO]

Problema 1.7 Un número capicúa es aquel que no cambia cuando se escribe al revés, por ejemplo 34143 y 1221 son capicúa, pero 145 no lo es porque $145 \neq 541$. Tampoco 30 es capicúa.

Determina cuantos números son capicúas entre L y R .

Enlace: [TODO]

Ejemplo

Para $L = 1$ y $R = 50$ la respuesta es 13.

Límites

$$1 \leq 1 \leq L \leq R \leq 10^5$$

www.omegaup.com/arena/problem/Contar-capicuas

Problema 1.8 Dado L y R , encuentra cuantos números primos¹ hay entre L y R , incluyendo L y R .

Entrada

Dos enteros L y R .

Salida

La cantidad de números primos en el rango.

Caso ejemplo

Entrada	Salida	Explicación
1 7	4	Los primos contados son: 2, 3, 5 y 7

Subtareas

- (50 pts) $1 \leq L \leq R \leq 10^3$
- (50 pts) $1 \leq L \leq R \leq 10^5$

www.omegaup.com/arena/problem/Cuenta-primos

¹Un número es primo si y solo si tiene exactamente dos divisores positivos, el 1 y el mismo.

Búsqueda completa

La búsqueda completa, también llamada fuerza bruta, es una técnica donde revisamos todos los posibles candidatos donde podría estar el o los valores que buscamos.

Esta búsqueda completa tiende a ser lenta, muchas veces incluso tiene complejidad exponencial y por esto, suele no ser la solución para los 100 puntos. Sin embargo, es muy útil conocerla ya que la fuerza bruta es fácil de pensar y siempre encuentra la respuesta si esta existe.

Ademas, la fuerza bruta suele resolvernros una o dos subtareas, dando un puntaje parcial. Esto es una forma de garantizar puntos en problema donde no se nos ocurra ideas mejores-

También hay muchos problemas que consisten en empezar de la fuerza bruta e ir mejorando de allí hasta que sea suficientemente buena.

No solo eso, además llega a ser útil para encontrar patrones y condiciones en las respuestas que no nos percatemos viendo la redacción, o para encontrar errores en un código que hayas hecho.

Las búsquedas completas suelen buscar la respuesta en:

- Los elementos de un arreglo (búsqueda lineal)
- Los valores de un rango (búsqueda lineal)
- Las parejas

- Los ordenes o permutaciones
- Los subconjuntos
- Las cadenas de decisiones

Probablemente reconozcas la búsqueda lineal, y esto es resultado de que esta es la búsqueda completa más sencilla en la que revisamos de forma secuencial un rango donde se puede encontrar la respuesta. Pero búsqueda completa incluye más tipos de iteraciones que solo revisar los valores de un ciclo.

Veamos a continuación cada una de estos casos y aprendamos a trabajar con ellos.

Pares 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 2.1.1

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 que 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 subtarear.

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 $(N - 1) + (N - 2) + \dots + 1 + 0$ iteraciones de j . Entonces hacemos $1 + 2 + 3 + \dots + (N - 1)$ iteraciones.

Usando la formula se suma de gauss² obtenemos que:

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

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 $\sim 10^4$.

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

Problemas de práctica

Problema 2.1.1 Definimos una inversión como una parejas (i, j) en un arreglo tal que $1 \leq i < j \leq N$ y también $A_i > A_j$. Es decir, una pareja de números que estén al revés de como deberían estarlo en un orden de menor a mayor.

Dado un arreglo de N enteros, imprime cuantas inversiones hay en él.

Ejemplo

Entrada	Salida	Expiación
4 3 2 6 1	4	Las inversiones son: El 3 con el 2, el 3 con el 1, el 2 con el 1, y el 6 con el 1.

Límites

- $1 \leq N \leq 5000$
- $1 \leq A_i \leq 10^9$

Enlace: [TODO]

Problema 2.1.2 Fernando regreso el otro día a la tienda de tornillos, y esta vez quiere darle una puntuación en un sitio web de guías locales. Fernando juzga la calidad de la tienda en función de cuantas cantidades diferentes de tornillos él puede comprar.

Recordemos que la tienda vende N cajas de tornillos, cada una con C_i dentro. Recordemos que Fernando solo puede tomar una o dos cajas en una compra porque solo tiene dos manos.

Dado las cajas de tornillos, determina la calidad de la tienda.

Ejemplo

Entrada	Salida	Expliación
3 1 3 4	5	Fer puede hacer compras de 1, 3, 4, 5 o 7 tornillos.

Límites

- $1 \leq N \leq 10^5$
- $1 \leq A_i \leq 10^3$

Enlace: [TODO]

Problema 2.1.3 La revista “algofashion” dijo esta semana que los números a la moda son aquellos que pueden ser representados como la suma de dos números pertenecientes a la secuencia de Fibonacci.

Recordemos que la secuencia de Fibonacci empieza con dos 1. Y luego cada número será resultado de la suma de los dos anteriores.

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

De forma que los primeros números de la secuencia son:

$$1, 1, 2, 3, 5, 8, 13, 21 \dots$$

Karel acaba de leer la revista y ahora quiere responder T preguntas, cada pregunta será del tipo: ¿El número x_i está de moda?

Como amigo, debes hacer un código que responda las dudas de Karel.

Entrada

En la primera línea vendrá el valor de T , cuantas preguntas tiene Karel.

En las siguientes T líneas vendrán las preguntas de Karel, una por línea. Cada pregunta consiste en un solo entero x_i .

Salida

Imprime T líneas, cada una siendo la respuesta a una pregunta de Karel. La línea i debe ser “SI” si x_i está a la moda y debe ser “NO” si no está a la moda.

Caso ejemplo

Entrada	Salida
3	SI
5	NO
6	SI
10	

Límites

- $1 \leq T \leq 100$
- $1 \leq x_i \leq 10^{18}$

Enlace: [TODO]

Problema 2.1.4 Tú tiene una constructora que busca comprar un terreno para construir un fraccionamiento.

Actualmente, el gobierno de Karelópolis permite que compres un terreno en la región para desarrollo. La región para desarrollo se ve como una cuadrícula de N filas y M columnas. El terreno que compres debe ser rectangular y alineada a la cuadrícula, es decir, no puedes comprar una celda de la cuadrícula de forma incompleta.

Además, tu calculaste cuanto dinero obtendrías si compras cada celda, en concreto, la celda que esta en la fila i y columna j aportaría v_{ij} pesos. Como hay celdas con valor positivo y otras con valor negativo, te preguntas cual es el máximo valor que puedes obtener.

Por ejemplo, si la región para desarrollo se ve de la siguiente forma:

1	-100	1	2
-3	5	2	-4
10	6	-1	4
-105	5	2	-100

Tu puedes obtener hasta 19 pesos comprando el terreno:

1	-100	1	2
-3	5	2	-4
10	6	-1	4
-105	5	2	-100

Encuentra el máximo valor posible si compras un terreno dada la región para desarrollo.

Entrada

En la primera línea recibes N y M .

En las siguientes N líneas recibirás M enteros representando los valores de v_{ij} .

Salida

Un entero que sea el mayor valor de un terreno.

Ejemplo

Entrada	Salida
4 4 1 -100 1 2 -3 5 2 -4 10 6 -1 4 -105 5 2 -100	19

Límites

- $1 \leq N, M \leq 75$
- $-10^9 \leq v_{ij} \leq 10^9$

Subtareas

- (40 pts) $1 \leq N, M \leq 15$
- (60 pts) Sin consideraciones extra.

Enlace: [TODO]

Problema 2.1.5 Dado un arreglo A de N enteros. Determina si existe un par de elementos que su suma modulo P sea K .

Límites

- $1 \leq N, K, P \leq 10^5$
- $1 \leq A_i \leq 10^9$

Subtareas

- (25 puntos) $1 \leq N \leq 10^3$
- (50 puntos) $1 \leq K, P \leq 10^3$
- (25 puntos) Sin restricciones extra.

Enlace: [TODO]

Subconjuntos

Antes de aprender a buscar subconjuntos, comencemos definiendo que son. Una vez sepamos que son, veremos como resolver problemas con ellos.

Definición de 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

Buscar 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:

Ejemplo 2.2.1

Karel tiene un conjunto formado por las primeras N letras del alfabeto. Ahora Karel que imprima 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, iterar por subconjuntos

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

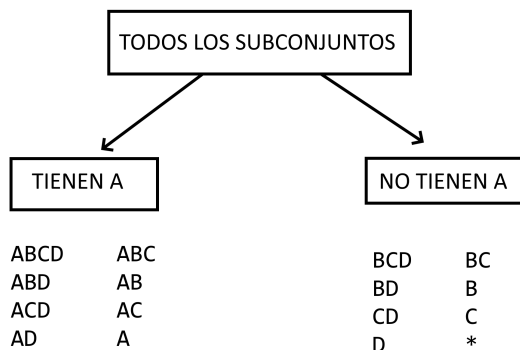
Digamos que $S(\text{conjunto})$ sean todos los subconjuntos del *conjunto*. Por ejemplo:

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

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 $S(ABCD) = A \rightarrow S(BCD) \cup S(BCD)$, donde $A \rightarrow S(BCD)$ 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 construya los subconjuntos usando las letras desde pos hasta $N - 1$.

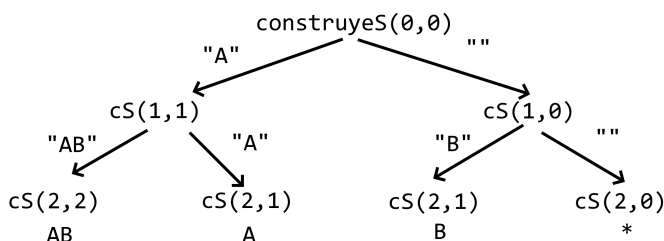
```
#include <iostream>
using namespace std;
int N;
char subconjunto[25];

// 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:



Un consejo para entender el código es simular la ejecución del programa paso a paso a mano en papel. Esto es útil cualquier otro algoritmo u técnica nueva.

Complejidad

La 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 2.2.2

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	Explicación
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:

```
#include<iostream>
#include<stdlib.h>
using namespace std;
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.
            Imprimos SI y terminemos el programa */
            cout << "SI";
            exit(0);
        }
        return;
    }
    buscaSubconjunto(pos+1, suma+Cajas[pos]); //Incluimos pos
        en el subconjunto
    buscaSubconjunto(pos+1, suma); //Excluimos 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;
}
```

Problemas de práctica

Problema 2.2.1 Warel se infiltró en una joyería con una mochila de capacidad W .

En esta joyería hay N diamantes, cada uno tiene un valor de v_i y un peso de w_i .

Warel puede robar cuantos diamantes quiera, pero el peso total de lo que se lleve no puede exceder a la capacidad de su mochila.

Determina la máxima cantidad de valor que Warel puede robar.

Entrada

En la primera línea viene el valor de N y W .

En la segunda línea vendrán los valores de los diamantes.

En la tercera y última línea vendrán los pesos de los diamantes.

Salida

La máxima suma de valor que Warel puede robar.

Caso Ejemplo

Entrada	Salida	Expliación
5 6 8 3 6 1 3 5 1 3 2 6	10	Toma el segundo, tercer y cuarto diamante. Tendrás valor de $10 = 3 + 6 + 1$ con peso $6 = 1 + 3 + 2$

Límites

- $1 \leq N \leq 20$
- $1 \leq W \leq 10^9$
- $1 \leq v_i, w_i \leq 10^9$

Enlace: [TODO]

Problema 2.2.2 Imprime todos los números binarios de N bits.

Imprime los números de menor a mayor y agrega 0s a la izquierda para que todos los números tengan la cantidad correcta de bits.

Ejemplo

Entrada	Salida
3	000
	001
	010
	011
	100
	101
	110
	111

Límites

- $1 \leq N \leq 20$

Problema 2.2.3 Un día encuentras una maquina que te permite invertir horizontalmente o verticalmente una matriz cuantas veces quieras.

Invertir verticalmente significa intercambiar la primera y ultima fila, la segunda y penúltima fila, la tercera y antepenúltima fila, etc. Mientras que invertir horizontalmente es lo mismo, pero intercambias las columnas en vez de las filas.

Se te dan dos matrices A y B , ambas de tamaño $N \times M$. Determina si la maquina te permite convertir la matriz A en la matriz B .

Entrada

Los valores de N y M .

Las siguientes N filas tendrán M enteros cada una. Siendo la matriz A .

Y otra vez seguirán N filas con M enteros cada una. Los valores de la matriz B .

Salida

Imprime "SI" en caso de que la matriz A pueda convertirse en la matriz B .

Ejemplos

Entrada	Salida
2 3 1 2 3 4 5 6 6 5 4 3 2 1	SI
2 2 1 2 3 1 2 3 1 1	NO

Límites

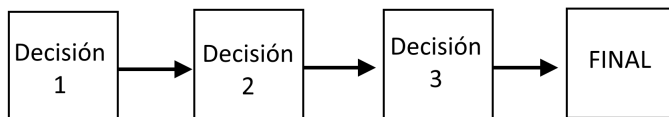
- $1 \leq N, M \leq 500$
- $1 \leq A_{ij}, B_{ij} \leq 10^9$

Enlace: [TODO]

Decisiones

Esta búsqueda es la verdadera búsqueda completa, la búsqueda exhaustiva. Es como la fórmula general de cualquier fuerza bruta.

La idea principal es que vamos a ver si se puede encontrar una solución que cumpla algo que pida el problema. Lo especial es que esta solución será resultado de una cadena de decisiones tomadas. Es decir, se pueden tomar decisión que hagan que se cumpla algo.



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

Es decir, 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?" y luego revisa "¿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 siguientes dos

formas:

```

busqueda(decision, solucion) {
    si (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] );
    }
}

```

Esta primera forma de hacer la búsqueda completa le llamamos backtrack-ing, que significa retroceder en inglés.

Esto es porque toma una opción haciendo recursión y luego cuando hace el return o termina la función, regresa en la pila recursiva, hacer un retroceso de su decisión.

Pero a veces conviene hacer otro tipo de búsqueda, dependiendo del tipo de decisiones a tomar, por ejemplo:

```

for (decision 1) {
    for (decision 2) {
        ...
        for (decision n) {
            revisar solucion
        }
    }
}

```

Esta es una idea abstracta y un poco extraña, pero es poderosa, de hecho es una forma de realizar cualquier búsqueda completa, incluyendo las búsquedas en pares, subconjuntos y ordenes.

Como siempre, ver un ejemplo ayuda mucho a entenderlo. Veamos un problema con el que ya estemos familiarizados.

Ejemplo 2.3.1

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

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

Fuente: TODO

Solución

Este problema lo vimos en la sección de iterar por pares, y eso es lo que debemos hacer:

```
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 .

De otra forma. Podemos ver que tenemos dos decisiones, primero decidir el valor de i y luego cuanto valdrá j . La primera decisión la resolvemos con el primer ciclo for y la segunda decisión con el otro for.

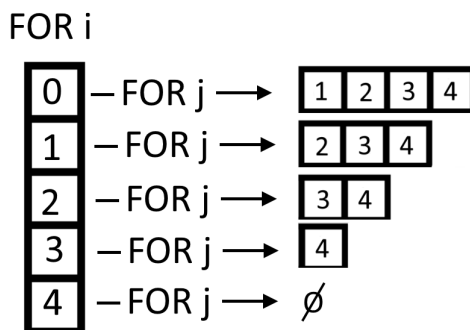
Visto con un código más similar al de esta búsqueda sería:

```
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]=j;//agregar j a la solucion
            buscar(decision+1);
            solucion[decision]=-1;//quitar j.
        }
    }
}
```

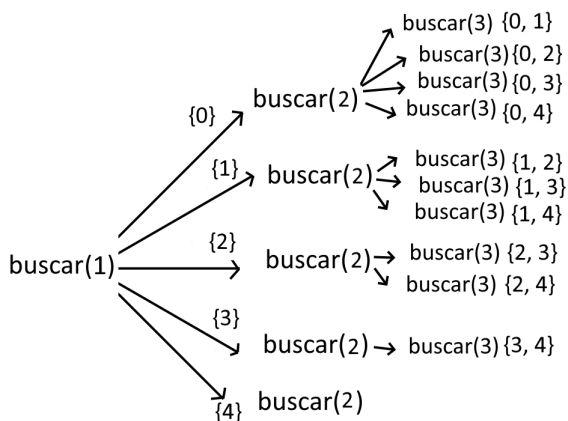
Estos dos códigos siguen la misma idea de obtener los pares fijando primero un valor de i , y luego fijando un valor para j ; solo lo realizan de una forma diferente.

Entiende porque los dos códigos son iguales 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:



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



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

Ejemplo 2.3.2

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, determina la repartición.

Entrada

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

La segunda línea tendrá n enteros a_1, a_2, \dots, 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

- $1 \leq N \leq 15$
- $1 \leq a_i \leq 10^6$

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";
}
```


Complejidad

La complejidad de una búsqueda exhaustiva varia mucho dependiendo de que tipo de solución se esta buscando, en el Ejemplo 2.3.1 vimos un ejemplo de complejidad $O(N^2)$.

Pero la complejidad del ejemplo anterior es $O(3^N)$. Esto es porque la complejidad de la búsqueda exhaustiva o completa es $O(\text{candidatos})$ 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.

Problemas de práctica

Problema 2.3.1 Estas en una torre la cual tiene N pisos.

El piso i tendrá K_i pasadizos que le permiten ir a pisos superiores. El único piso sin pasadizos es el último, el piso N .

Tu quieres ver cuantas formas hay de ir del piso 1 al N . Dos maneras de ir del piso 1 al N son diferentes si en una usas un pasadizo y en la otra no.

Nota: esta prohibido tomar los pasadizos de un piso superior a uno inferior

Entrada

Dos enteros N , la cantidad de pisos en la torre.

En las siguientes $N - 1$ lineas vendrá la descripción de cada piso.

Cada piso constará de un entero K_i representando cuantos pasadizos hay allí que conectan hacia arriba. Seguido, vendrán K_i enteros diferentes, representando a que pisos tienes un pasadizo. Solo podrás ir a pisos superiores.

Salida

Imprime un entero que represente la cantidad de formas de ir del piso 1 al piso N .

Ejemplo

Entrada	Salida	Expiación
4	3	Tienes tres formas de llegar:
2 2 3		$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
2 3 4		$1 \rightarrow 2 \rightarrow 4$
1 4		$1 \rightarrow 3 \rightarrow 4$

Límites

- $1 \leq N \leq 16$
- $1 \leq K_i < N$

www.omegaup.com/arena/problem/ TODO

Problema 2.3.2 El COMI diseñó unos mapas futuristas. Los mapas futuristas son como los mapas de ahora, pero están impresos en una superficie transparente. Ellos tienen un archivo de varios mapas de la galaxia. Sebastian y Héctor se dieron cuenta de que existen varios mapas repetidos, por lo que desean eliminar cualquier repetición. Para esto, Sebastian toma un mapa, Héctor toma otro y ambos comparan los mapas para determinar si son iguales. Sin embargo, Héctor estaba distraído y algunos mapas se los pasaba en una posición diferente a la original. Ahora necesitan tu ayuda para comparar los dos mapas y saber si son iguales.

Un mapa está representado por una matriz cuadrada de Xs y 0s. Dos mapas son iguales si ambos tienen el mismo carácter en la misma coordenada, para todas las coordenadas.

Para validar que dos mapas son iguales, puedes aplicar cualquiera de las siguientes acciones cero o más veces sobre un mapa que quieras comparar con otro:

- Rotarlo 90° .
- Rotarlo 180° .
- Rotarlo 270° .
- Invertirlo horizontalmente.
- Invertirlo verticalmente.

X	0	0
0	X	X
0	X	0

Mapa Original

0	0	X
X	X	0
0	X	0

Rotar 90°

0	X	0
X	X	0
0	0	X

Rotar 180°

0	X	0
0	X	X
X	0	0

Rotar 270°

0	X	0
0	X	X
X	0	0

**Invertir
Verticalmente**

0	0	X
X	X	0
0	X	0

**Invertir
Horizontalmente**

Problema

Se te mostrará dos mapas y debes ayudarles a decir si son iguales o diferentes. Si son iguales, deberás escribir **IGUALES**. Si son diferentes, deberás escribir **DIFERENTES**.

Entrada

En la primera línea N , la longitud del lado de cada mapa. En las siguientes N líneas una cadena de N caracteres **0** ó **X** que representan el primer mapa. En las siguientes N líneas una cadena de N caracteres **0** ó **X** que representan el segundo mapa.

Salida

Debes escribir **IGUALES** si los mapas son iguales después de hacer todas las transformaciones necesarias o **DIFERENTES** en caso contrario.

Ejemplo

Entrada	Salida	Expiación
4 XOOO XXOO OOOO XXXX XOOO XOOO XOXO XOXX	IGUALES	Si el segundo mapa lo giras 90^0 y además lo inviertes verticalmente, obtienes la misma distribución que el primer mapa.
2 XX OO XO OX	DIFERENTES	No hay manera de transformar el segundo mapa para que se vea tal como el primero.
4 XOOO XXOO OOOO XXXX XOOO XXOO OOOO XXXX	IGUALES	En este caso los mapas ya son iguales, sin necesidad de aplicar ninguna operación.

Límites

$$1 \leq N \leq 500$$

Subtareas

- (25 puntos)
 - $1 \leq N \leq 10$
 - Se asegura que, en caso de que los mapas sean iguales, no es

necesario aplicar ninguna acción.

- (25 puntos)
 - $1 \leq N \leq 50$
 - Se asegura que, en caso de que los mapas sean iguales, no es necesario aplicar ninguna rotación.
- (25 puntos)
 - $1 \leq N \leq 50$
 - Se asegura que, en caso de que los mapas sean iguales, no es necesario aplicar ninguna inversión vertical o inversión horizontal.
- (25 puntos)
 - $1 \leq N \leq 500$
 - Cualquier acción podría ser necesaria para validar que los mapas son iguales.

Fuente: **OMI 2020**

www.omegaup.com/arena/problem/OMI-2020-Mapas

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 completa tardaría miles de millones de años en menos de un segundo. Y esta impresionante herramienta ahora será tuya.

La búsqueda binaria difiere de la completa 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 completa va eliminando opciones de una en una.

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

Número de pasos	Candidatos restantes	
	Completa	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 $\lceil \log_2(\text{candidatos}) \rceil$ pasos ³. 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.

³ $\lceil \log_2(A) \rceil$ 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 3.1

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.

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

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 formulas del cuadrado.

$$\text{Perímetro} = 4 \times \text{Lado}$$

$$\text{Área} = \text{Lado} \times \text{Lado}$$

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 corre para la subtask 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 $(5 \times 10^8) \times (5 \times 10^8) \geq A$, 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 long 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 $L=\text{buscar}(1, 1000000000, A)$.

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 $L=\text{sqrt}(A)$ con la librería `<math.h>` que calcula el valor rápidamente, pero se uso búsqueda binaria para ejemplificar.

Ejemplo 3.2

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

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 índice 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 $O(\log N)$ por pregunta, en total $O(T \log N)$

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 que las que no se puede 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.

Problemas de práctica

Problema 3.1 Fernando es el operador de montacargas en un almacén cuando le pidieron que sacara K piezas del inventario.

Sin embargo, las piezas están organizadas en una pila de N cajas. La caja i contiene C_i piezas.

Como funciona el montacargas, Fer puede agarrar las primeras R cajas de la pila y traerlas consigo.

Es decir, si $C = [1, 2, 3, 4]$ y $R = 2$, el montacargas se llevará las cajas $[1, 2]$.

Como el montacargas gasta gasolina según cuantas cajas cargue y esta es muy cara, ayuda a Fer a determinar la mínima R para Q valores de K .

Entrada

Un entero N representando la cantidad de cajas.

En la siguiente línea habrá N enteros, los valores de C_1, C_2, \dots, C_N .

En la tercera línea estará el entero Q , para cuantos valores debes descubrir R .

En las siguientes Q líneas habrá un entero K que representa la cantidad de piezas deseada, suponiendo que siempre tenemos las mismas N cajas.

Salida

Por cada una de las Q preguntas: imprime una línea con la mínima R para obtener K piezas, si no se puede imprime -1 .

Ejemplo

Entrada	Salida	Expliación
3	1	Agarrando la caja No. 1 obtenemos 1 pieza
1 4 1	2	
3	-1	Tomando la caja No. 1 y No. 2, tenemos 5 piezas, las cuales son mas que 3.
1		
3		
10		No hay suficientes piezas para 10.
5	2	
3 1 1 3 1		
1		
4		

Límites

- Para 50%
 - $0 < N \leq 10^2$
 - $0 < Q \leq 10^3$
- Para 100%
 - $0 < N \leq 10^5$
 - $0 < Q \leq 10^5$
 - $0 < C_i \leq 1000$
 - $0 \leq K \leq 10^9$

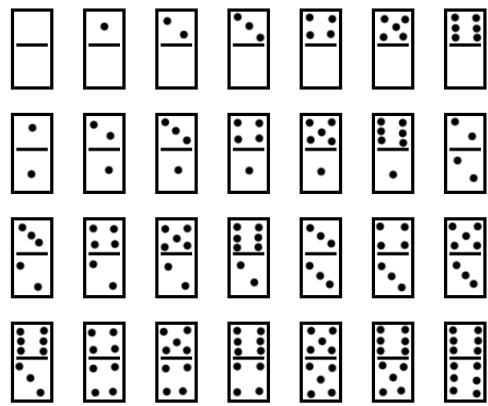
Enlace: www.omegaup.com/arena/problem/Transporte-de-Cajas

Problema 3.2 Sho colecciona juegos de dominó y quiere jugar.

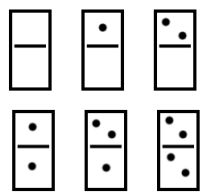
Pero los dominós de Sho son un poco especiales, ya que son k -dominós. El dominó tradicional es del tipo 6-dominó, porque tiene números del 0 al 6, con un total de 28 fichas.

Entonces, un k -dominó tendrá números del 0 al k . Con una ficha por cada par posible de números, incluyendo las mulas (mula es el mismo número emparejado consigo mismo).

Las fichas del 6-dominó son:



Y las del 2-domino son las siguientes seis:



Sho quiere jugar con al menos N fichas, pero como después de jugar tiene que guardar, quiere usar el dominó con menor cantidad de fichas que tenga por lo menos N fichas.

Dado N , encuentra el valor de k para el k -dominó que le sirve a Sho.

Ejemplo

Entrada	Salida
30	6
6	2
1000	44

Límites

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

Enlace: TODO

Problema 3.3 Adivina el número que esta pensando OmegaUp.

Deberás implementar una función `void adivina(long long a, long long b)` que descubra el número s que OmegaUp esta pensando. Se cumplirá que $a \leq s \leq b$.

Para adivinar el número podrás llamar a la función `long long pista(long long x)`.

Esta función regresará:

- -1 si el número que piensa OmegaUp es menor que x .
- 0 si el número que piensa OmegaUp es x .
- 1 si el número que piensa OmegaUp es mayor que x .

La ultima llamada a pista debe ser con el número que OmegaUp esta pensando.

Límites

- $-2^{61} \leq a, b \leq 2^{61}$

Evaluación

Tu puntuación será en base a la cantidad de llamadas que hagas a la función `long long pista(long long x)` de la siguiente manera:

- 100% si haces a lo más $\log(b - a + 1)$ llamadas
- 50% si haces a lo más $2\log(b - a + 1)$ llamadas
- 0% si haces más de $2\log(b - a + 1)$ llamadas

Enlace: www.omegaup.com/arena/problem/COMI-Adivina-el-numero

NOTA: Este es un problema interactivo, si no conoces como trabajar con estos, ve la página: 113

Problema 3.4 A.R.C. Markland-N es un edificio alto con n pisos enumerados del 1 al n . Entre cada dos pisos adyacentes existe una escalera que los conecta.

Es hora del almuerzo para nuestro sensei, Colin "ConneR" Neumann Jr, y él está planeando la ubicación donde disfrutar su comida.

La oficina de ConneR esta en el piso s del edificio. En cada piso (incluyendo s) hay un restaurante ofreciendo comida. Sin embargo, debido a renovaciones que están en proceso, k restaurantes se encuentran cerrados y por lo tanto, ConneR no puede disfrutar su almuerzo en ellos.

ConneR quiere llegar a un restaurante tan rápido como sea posible. ¿Cuál es el mínimo número de escaleras que debe usar para llegar al restaurante más cercano abierto?

Entrada

En la primera línea habrá un entero t – El número de casos pruebas. Luego vendrán las descripciones de cada uno de los t casos de prueba.

La primera línea de un caso de prueba tiene tres enteros n , s y k – Respectivamente, el número de pisos, el piso donde está ConneR y el número de restaurantes cerrados.

La segunda línea del caso de prueba tendrá k enteros distintos $a_1, a_2 \dots a_k$ – Los números de piso con su restaurante cerrado.

Salida

Por cada caso de prueba imprime una línea con la respuesta – el mínimo número de escaleras que ConneR debe usar para llegar a un restaurante abierto.

Ejemplo

Entrada	Salida
5	2
5 2 3	0
1 2 3	4
4 3 3	0
4 1 2	2
10 2 6	
1 2 3 4 5 7	
2 1 1	
2	
100 76 8	
76 75 36 67 41 74 10 77	

Límites

- $1 \leq t \leq 1000$
- $2 \leq n \leq 10^9$
- $1 \leq s \leq n$
- $1 \leq k \leq \min(n - 1, 1000)$
- Se garantiza que la suma de los valores de k no excederá 1000.

Fuente: codeforces.com

www.codeforces.com/problemset/problem/1293/A

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 3.3

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 almacenar 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.

Enlace: [TODO]

Solución

Este problema 1.6 de búsqueda lineal con validación en la página 13, 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 y R . 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 $\text{funciona}(m)$ cumple, también lo hará cualquier valor mayor que m .
- Si $\text{funciona}(m)$ falla, también lo hará cualquier valor menor que m .

Es decir, se ve de la siguiente forma:

TODO PONER IMAGEN AQUI

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;
```

Ahora, recordemos que la complejidad de la búsqueda binaria es $O(\log R)$, pero en cada paso de la búsqueda estamos llamando a $\text{funciona}()$ que tiene complejidad $O(N)$, por lo tanto, la complejidad total es $O(N \log R)$.

Problemas de práctica

Problema 3.5 Barcos turísticos

El COMI ha preparado un paseo por barco para los olimpicos del OMI por el río *algorrio*.

El concurso está organizado por M delegaciones enumeradas del 1 al M , cada una con a_i olímpicos.

Para este paseo se planean contratar K barcos en el que se subirán los olímpicos. Para que todo sea seguro, se debe poner un límite X de forma de que en un barco no pueda haber más de X olímpicos a bordo.

Para el abordaje, cada delegación se subirá a un barco de forma que todos los olímpicos de la delegación estén en el mismo barco. Además, si en un barco está la delegación x y la delegación y tal que $x < y$, entonces también debe estar la delegación z en el mismo barco si se cumple que $x < z < y$. En otras palabras, los barcos tendrán un rango consecutivo de delegaciones.

¿Cuál es la mínima X que permita que todas las delegaciones se suban a los barcos?

Entrada

La primera línea tiene dos enteros M y K –la cantidad de delegaciones y barcos.

La segunda línea tendrá M enteros a_1, a_2, \dots, a_M – a_i es la cantidad de olímpicos en la delegación i .

Salida

Imprime un entero – El menor valor de X que permita que todos los olímpicos se suban a algún barco.

Ejemplo

Entrada	Salida	Explicación
5 3 7 5 8 5 9	13	Los barcos serán 1. La delegación 1 y 2 2. La delegación 3 y 4 3. La delegación 5

Límites

- $1 \leq N, K \leq 10^5$
- $1 \leq a_i \leq 10^9$

TODO: www.omegaup.com/arena/problem/arg1

Problema 3.6 Dado un arreglo A de N enteros, determina si existe un subcadena que sume K .

Una subcadena es cualquier subarreglo de elementos continuos. Para el arreglo $[1, 2, 10, 5, 3]$, tenemos 15 subcadenas, algunas de estas son: $[1, 2, 10]$, $[10, 5]$, $[2, 10, 5]$, $[3]$.

- $1 \leq N \leq 10^5$
- $1 \leq A_i, K \leq 10^9$

Búsqueda en los reales

Hay veces en la que no estamos buscando un valor entero, si no en vez queremos 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.

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 3.5

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 3.1, 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;  
}
```

Problemas de practica

Problema 3.7 Fernando ha adquirido un gusto por las carreras de coches en la fórmula π .

En este deporte, los coches recorren una pista recta que mide L metros de longitud. Además, dependiendo de resultados anteriores el coche i inicia con x_i metros de ventaja.

Como Fer es un aficionado muy dedicado, se ha percatado de que cada coche se moverá hacia la meta con una velocidad constante, en concreto, el carro i tendrá se moverá a v_i metros por segundo.

Fernando se pregunta, en que tiempo (medido en segundos) cruza el coche que termina en lugar K .

Entrada

Tres enteros N , L y K , la cantidad de coches en la carrera, la longitud de la pista y el lugar que te interesa.

En la segunda línea habrá N enteros, x_1, x_2, \dots, x_n , donde x_i es la posición inicial del coche i .

En la tercera línea vendrán N enteros, v_1, v_2, \dots, v_n , donde v_i es la velocidad del coche i .

Salida

Un solo número, indicando el segundo en el cual pasa la meta el coche que termina en lugar K . Con precisión absoluta de menor o igual que 10^{-6} .

Ejemplos

Entrada	Salida
4 10 2 1 2 3 4 3 3 1 10	2.666667

Límites

- $1 \leq N, L, K \leq 10^5$
- $1 \leq v_i \leq 10^3$
- $0 \leq x_i < L$

Problema 3.8 Planetas:

www.omegaup.com/arena/problem/planetas

Problema 3.9 Foto

www.omegaup.com/arena/problem/carretera

Problema 3.10 La siguiente lección en preparatoria requiere que dos temas sean discutidos. El i -ésimo tema es interesante por a_i unidades para el profesor y b_i unidades para los estudiantes.

Un par de temas i y j ($i < j$) es llamado **bueno** si $a_i + a_j > b_i + b_j$ (es decir, es más interesante para el profesor).

Tu tarea es encontrar el número de parejas de temas **buenas**.

Entrada

Un entero n , la cantidad de temas.

La segunda línea tiene n enteros: a_1, a_2, \dots, a_n , donde a_i es cuan interesante es el tema i para el profesor.

La tercera línea tiene n enteros: b_1, b_2, \dots, b_n , donde b_i es cuan interesante es el tema i para los estudiantes.

Salida

Un entero, la cantidad de parejas buenas.

Ejemplo

Entrada	Salida
5 4 8 2 6 2 4 5 4 1 3	7
4 1 3 2 4 1 3 2 4	0

Límites

- $2 \leq N \leq 2 \cdot 10^5$
- $1 \leq a_i, b_i \leq 10^9$

Fuente: codeforces.com

www.codeforces.com/problemset/problem/1324/D

Binaria en C++

En C++ hay algunas funciones ya programadas que pueden hacer búsqueda binaria por nosotros.

Principalmente reconocemos tres de estas, `binary_search`, `lower_bound` y `upper_bound`.

Todas estas funciones requieren incluir `<algorithm>`

Binary search

Nos permite saber si un elemento concreto se encuentra o no en un arreglo ordenado.

A continuación se muestra un ejemplo de un arreglo con N elementos.

```
if (binary_search(arreglo, arreglo+N, elemento)) {  
    cout << "Existe: "<<elemento<<" en el arreglo";  
} else {  
    cout << "No existe: "<<elemento<<" en el arreglo";  
}
```

Lower bound

Encuentra el primer elemento menor o igual que un valor en un arreglo ordenado. Si el elemento no existe, nos regresa un iterador al final.

Veamos como obtener el indice del elemento encontrado.

Con arreglos:

```
int posicion =  
    lower_bound(arreglo, arreglo+N, valor)-arreglo;
```

Con un vector queda como:

```
int posicion =
```

```
| lower_bound(arr.begin(), arr.end(), valor)-arr.begin();
```

Si no hay elemento mayor o igual en el arreglo, obtenemos `posicion=N`.

Lo que en realidad regresa `lower_bound`, es un iterador, si te interesa más del tema, puedes leer sobre ellos en internet, para nosotros nos basta con saber la formula de arriba.

Upper bound

Encuentra el primer elemento estrictamente mayor que un valor en un arreglo ordenado.

Podemos usarlo como `lower_bound`:

Con arreglos:

```
| int posicion =  
|   upper_bound(arreglo, arreglo+N, valor)-arreglo;
```

Con un vector queda como:

```
| int posicion =  
|   upper_bound(arr.begin(), arr.end(), valor)-arr.begin();
```

Obtendremos `posicion = N` si no existe elemento estrictamente mayor que valor.

Búsqueda en profundidad

La búsqueda en profundidad, o DFS por sus siglas en ingles, es una búsqueda para ver si existe una solución tomando una serie de operaciones o decisiones. Es muy similar a la búsqueda completa que vimos en la pagina 41, pero es una versión optimizada de esa técnica.

La idea es ir explorando todas las decisiones, pero si una serie de decisiones nos llevan al mismo lugar, ya no volver a explorar por allá. Es decir, la diferencia entre la búsqueda exhaustiva y la DFS, es que esta si detecta que no necesita explorar un lugar que ya fue revisado previamente, no lo hace.

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 4.1

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);,

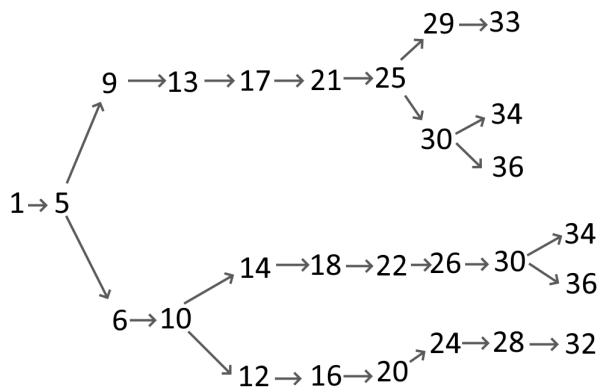
    //presiona el boton 2 si es posible.
    if (x%b==0) {
        solucion[paso]=2;
        exhaustiva(x+x/b, pasos+1);
    }
}

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.

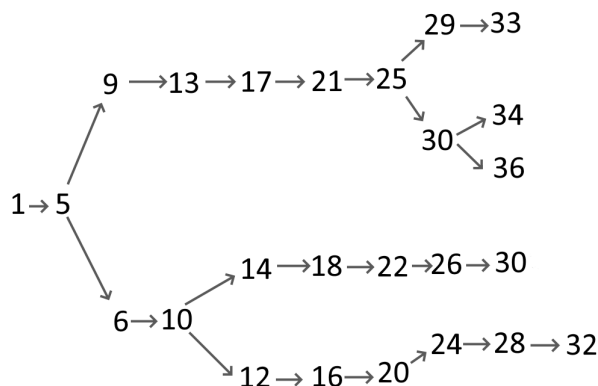


Allí vemos que podemos llegar al 30 con dos rutas. Pero sin importar cual ruta sigamos, de allí solo podemos llegar a los mismos números, al 34 y al 36.

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

No, no lo tiene. Al llegar por segunda vez podemos detener 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;
    }
    //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.

Estados y transiciones

Hasta ahorita este libro ha sido un poco informal en como le llama a "decisiones/lugares" que son alcanzables con las "opciones".

En el ejemplo 4.1 teníamos como "lugar", el número x y las "opciones" eran presionar el botón 1 o el botón 2.

Pero ahora vamos a ser un poco más formales y darles el nombre correcto y debido a esto.

Los lugares a los que llevamos, donde tenemos que tomar decisiones que los representamos con argumentos en la función recursiva, el nombre que usaremos ahora es estado. Es decir, es un estado de la búsqueda a la cual puedes llegar después de tomar varias "opciones".

Y a las "opciones" que tomamos, que nos mueven de un estado a otro y que son las llamadas recursivas que hacemos les pondremos de nombre formal "transiciones".

Entonces, nuestra búsqueda DFS explora todos los estados alcanzables desde uno inicial, utilizando las transiciones disponibles.

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 4.1 teníamos y estados, así como a lo más $2y$ transiciones (dos por cada estado de los botones), la complejidad del ejemplo es pues: $O(V + E) = O(y + 2y) = O(y)$.

Problemas de práctica

Problema 4.1 Te encuentras en una torre que tiene N pisos enumerados del 1 al N .

El piso i tiene P_i pasadizos, cada pasadizo te permitirá moverte a un piso superior desde i . Los pasadizos no se puede usar para bajar. Además, el último piso no tiene pasadizos.

Desgraciadamente, habrá veces en que te sea imposible viajar del piso 1 al piso j debido a como fueron construidos los pasadizos.

Ahora, dado la descripción de los pisos, determina a cuantos pisos puedes alcanzar desde el piso 1.

Entrada

Dos enteros N , la cantidad de pisos en la torre.

En las siguientes $N - 1$ líneas vendrá la descripción de cada piso.

Cada piso constará de un entero P_i representando cuantos pasadizos hay allí que conectan hacia arriba. Seguido, vendrán P_i enteros diferentes, representando a que pisos tienes un pasadizo. Solo habrá pasadizos a pisos superiores.

Salida

Imprime un entero que represente la cantidad de pisos que son alcanzables desde el primero.

Ejemplo

Entrada	Salida	Expliación
10 3 2 5 7 1 3 0 2 4 5 1 7 3 6 8 10 1 10 0 2 9 10 9 10	5	Podemos visitar los pisos: 1, 2, 3, 5 y 7.

Límites

- $1 \leq N \leq 5 \times 10^4$
- $0 \leq P_i \leq 10$

Subtareas

- (45 puntos) $1 \leq N \leq 12$
- (55 puntos) $1 \leq N \leq 5 \times 10^4$

ENLACE: TODO

Problema 4.2 Javier esta en un tablero de N filas y M columnas. En este tablero habrá unas celdas libres y unas celdas bloqueadas.

Él se encuentra en la casilla superior izquierda, la $(1, 1)$, y quiere llegar a la casilla (f, c) . La casilla donde Javier inicia siempre será libre.

Javier puede en cualquier momento, dar un paso y moverse a cualquiera de las cuatro casillas de arriba, abajo, izquierda o derecha. Siempre y cuando estas casillas existan y no estén bloqueadas.

Determina si existe un camino de donde esta Javier a donde quiere ir.

Entrada

Dos enteros N y M , el número de filas y el número de columnas del tablero.

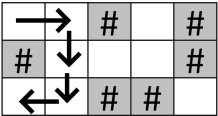
En las siguientes N líneas recibirás M caracteres, siendo la descripción del tablero. '#' simboliza una casilla bloqueada y '.' es una libre.

En la última línea tendrás dos enteros. f y c , la fila y columna donde esta la celda a la que Javier quiere ir.

Salida

Imprime SI en caso de que exista un camino de $(1,1)$ a (f,c) . De lo contrario imprime NO.

Ejemplo

Entrada	Salida	Expiación
3 5 ..#.# #...# ..##. 3 1	SI	
4 5# #...# #...# ...#. 3 5	NO	

Límites

- $2 \leq N, M \leq 1000$
- $1 \leq f \leq N$
- $1 \leq c \leq M$

www.omegaup.com/arena/problem/ TODO

Búsqueda en amplitud

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, y luego se regresa de esa exploración para ver la siguiente opción.

En cambio, la BFS lo que hace es llevar una lista de "estados por explorar" y procesarlos en orden. Para iniciar, revisamos todas las transiciones del estado inicial y los estados a los que nos lleven los agregamos en la lista.

TODO pon imagen aquí.

Y luego para cada una de esos estados los exploramos también, es decir, expandimos las transiciones de esos estados y lo nuevo que encontremos lo ponemos en la lista de "estados por explorar". Importante: los visitamos en el orden que fueron agregados a la lista por explorar.

TODO pon imagen aquí.

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 nos permite saber el mínimo número de transiciones necesarias para llegar e incluso lo podemos guardar en un arreglo o map para consultar después.

Para lograr este comportamiento, es importante ver que la lista por explorar se comporta de la siguiente forma: los estados son procesados en el mismo orden en el que entran, de forma que el primer estado en entrar a la lista es el primero en ser atendido. Este comportamiento es similar a la cola del cine, donde se atiende en el orden que llegas.

Para esto, podemos usar una estructura de datos de C++ llamada **queue**, que significa cola en ingles. Si no sabes trabajar con ella, revisa el anexo en la página 121.

Entonces, una BFS lo que hace es:

Tener una cola de la lista por explorar llamada **explorar**, un arreglo de booleanos **visitado** para marcar los estados que ya han sido descubiertos por la BFS y agregaremos un arreglo de enteros **no_transiciones** para guardar cuantas transiciones requerimos para llegar a cada estado.

Comenzamos agregando a la lista por explorar el estado inicial con un **no_transiciones** de 0. Luego, mientras todavía allá estados por explorar, lo procesaremos. Lo sacaremos de la pila al ya ser explorado y agregaremos a **explorar** todos los estados que no hayan sido visitados alcanzables desde el estado que estamos procesando.

En código:


```

bool visitado[];
int no_transiciones[];
void BFS(inicio) {
    queue<estados> explorar;
    cola.push(inicio);
    visitado[inicio]=true;
    no_transiciones[inicio]=0;
    while (explorar.empty()==false) {
        actual = explorar.front();
        cola.pop();
        if (actual es solucion) {
            registrar solucion actual;
            break; //opcional, si solo nos interesa una solucion
        }
        for (transiciones de actual) {
            if (!visitado[transicion]) {
                visitado[transicion]=true;
                no_transiciones[transicion]=
                    no_transiciones[actual]+1;
                explorar.push(transicion);
            }
        }
    }
}

```

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

Ejemplo 5.1

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	Explicación
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

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

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 no_transiciones[100005];
int bfs(int x, int y, int a, int b) {
    queue<int> cola;
    cola.push(x);
    visitado[x]=true;
    no_transiciones[x]=0;
    while (cola.empty()==false) {
        int actual=cola.front();
        cola.pop();
        if (actual==y) {
            return no_transiciones[y];
        }
        //Primer boton
        int siguiente = actual+a;
        if (visitado[siguiente]==false){
            visitado[siguiente]=true;
            no_transiciones[siguiente]=
                no_transiciones[actual]+1;
            cola.push(siguiente);
        }
        //Segundo boton
        siguiente = actual+actual/b;
        if (actual%b == 0 && visitado[siguiente]==false){
            visitado[siguiente]=true;
            no_transiciones[siguiente]=
                no_transiciones[actual]+1;
            cola.push(siguiente);
        }
    }
    return -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 $O(V + E) = O(y + 2y) = O(y)$.

Problemas de práctica

Problema 5.1 Vasya encontró un dispositivo extraño. En el panel frontal hay: un botón rojo, un botón azul y una pantalla mostrando un entero positivo.

Después de pulsar el botón rojo, el dispositivo multiplica por dos el número. Al pulsar el botón azul, el dispositivo le resta uno al número de la pantalla.

Si en algún momento el número deja de ser positivo, el dispositivo se rompe. La pantalla puede mostrar números arbitrariamente grandes. Inicialmente se muestra el número n .

Vasya quiere obtener el número m en la pantalla. ¿Cuál es la menor cantidad de pulsaciones que debe hacer para obtener este resultado?

Entrada

La primera y única línea de la entrada tiene dos enteros distintos n y m .

Salida

Imprime un solo número – La mínima cantidad de veces que uno debe presionar los botones para obtener m del número n .

Ejemplos

Entrada	Salida	Expliación
4 6	2	Pulsa una vez el botón azul y luego el rojo.
10 1	9	No necesitamos duplicar el número por lo que presionamos el azul nueve veces.

Límites

$$1 \leq n, m \leq 10^4$$

Fuente: codeforces.com

www.codeforces.com/problemset/problem/520/B

Problema 5.2 Se te dan dos números enteros, n y x . Tu puedes realizar múltiples operaciones al entero x .

En cada operación que hagas consiste en: Elige un dígito y que este en la representación decimal de x al menos una vez, y reemplazar x por $x \cdot y$.

Quieres hacer que la longitud de la representación decimal de x (sin ceros a la izquierda) sea igual a n . ¿Cuál es la menor cantidad de operaciones para lograr esto?

Entrada

La única línea de la entrada contiene dos enteros n y x .

Salida

Imprime un entero – El mínimo número de operaciones requeridas para hacer la longitud de la representación decimal de x (sin ceros a la izquierda) igual a n , o -1 si es imposible.

Entrada	Salida	Explicación
2 1	-1	
3 2	4	1. multiplica x por 2, tal que $x = 2 \cdot 2 = 4$; 2. multiplica x por 4, tal que $x = 4 \cdot 4 = 16$; 3. multiplica x por 6, tal que $x = 16 \cdot 6 = 96$; 4. multiplica x por 9, tal que $x = 96 \cdot 9 = 864$;
13 42	12	

Límites

- $2 \leq n \leq 19$
- $1 \leq x \leq 10^{n-1}$

Fuente: codeforces.com

www.codeforces.com/problemset/problem/1681/D

Problema 5.3 Javier esta en un tablero de N filas y M columnas. En este tablero habrá unas celdas libres y unas celdas bloqueadas.

Él se encuentra en la casilla superior izquierda, la $(1, 1)$, y quiere llegar a la casilla (f, c) . La casilla donde Javier inicia siempre será libre.

Javier puede en cualquier momento, dar un paso y moverse a cualquiera de las cuatro casillas de arriba, abajo, izquierda o derecha. Siempre y cuando estas casillas existan y no estén bloqueadas.

Determina la mínima cantidad de pasos para ir de $(1, 1)$ a (f, c) .

Entrada

Dos enteros N y M , el número de filas y el número de columnas del tablero.

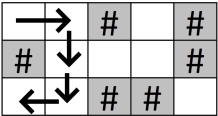
En las siguientes N líneas recibirás M caracteres, siendo la descripción del tablero. '#' simboliza una casilla bloqueada y '.' es una libre.

En la última línea tendrás dos enteros. f y c , la fila y columna donde esta la celda a la que Javier quiere ir.

Salida

Imprime un entero – La mínima cantidad de pasos para ir de $(1, 1)$ a (f, c) . Si no se puede imprime -1 .

Ejemplo

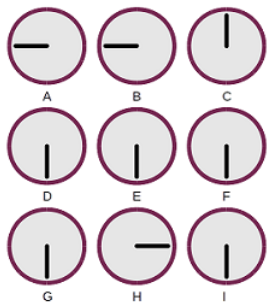
Entrada	Salida	Expliación
3 5 ..#.# #...# ..##. 3 1	4	
4 5# #...# #...# ...#. 3 5	-1	

Límites

- $2 \leq N, M \leq 1000$
- $1 \leq f \leq N$
- $1 \leq c \leq M$

TODO www.omegaup.com/arena/problem/

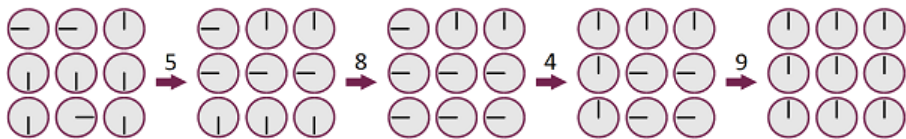
Problema 5.4 Hay nueve relojes dispuestos en un arreglo de 3×3 (véase la imagen), etiquetado con las las letras de la *A* a la *I*. El objetivo d este problema es regresar las manecillas de todos los relojes para que marquen las 12 en punto en el menor número de movimientos posibles.



Hay nueve diferentes movimientos permitidos para modificar las manecillas de los relojes. Cada movimiento permitido es identificado por un número del 1 al 9. Un movimiento afecta únicamente a un determinado conjunto de relojes; cada reloj afectado mueve su manecilla 90° (grados) en sentido horario. A continuación se encuentra la descripción de los relojes afectados por los nueve movimientos:

Movimiento	Relojes afectados
1	ABDE
2	ABC
3	BCEF
4	ADG
5	BDEFH
6	CFI
7	DEGH
8	GHI
9	EFHI

La siguiente figura muestra una posible solución para el caso presentado en la imagen de arriba, descrita por los números de movimiento 5, 8, 4 y 9. Nota que esta no es la única solución para el caso de ejemplo.



Entrada

res líneas con tres enteros cada una; cada entero representa la hora inicial de cada reloj, en el orden A, B, C, \dots, H, I . Los enteros tendrán un valor 3, 6, 9 o 12, representando la hora que marca un reloj.

Salida

Imprime en salida estándar una sola línea con una lista de enteros separados por espacios: la secuencia de movimientos más corta que pone todos los relojes marcando las 12 en punto. En caso de que existan múltiples secuencias válidas con longitud mínima, cualquiera de ellas será aceptada.

Ejemplo

Entrada	Salida
9 9 12	5 8 4 9
6 6 6	
6 3 6	

Fuente: IOI 1994

www.omegaup.com/arena/problem/relojes

Problemas

En esta sección encontrarás una lista de problemas que se resuelven con los temas vistos en este libro.

Problema E.1 La leyenda dice que el tesoro de Moctezuma está enterrado en el Centro Histórico de la Ciudad de México. El Centro Histórico está representado como una cuadrícula de n filas y m columnas. Gracias a la tecnología de la nueva app iFind puedes desenterrarlo por fin.

iFind es una app donde especificas una casilla (i, j) de la cuadrícula y la app responde cuántos tesoros hay enterrados en el área del rectángulo que abarca de la fila 1 a la fila i y de la columna 1 a la columna j .

Una vez que sabes la casilla exacta donde hay un tesoro debes cavar en esa posición para desenterrarlo.

Se asegura que cada casilla solo puede tener o tesoro y que no hay más de tesoro en cada columna.

Problema

Escribe un programa que dados n y m , el alto y ancho de la cuadrícula y k , el número total de tesoros enterrados, desentierre los k tesoros usando la menor cantidad posible de preguntas a la app.

Interacción

No necesitas leer o escribir⁴, debes implementar en tu código la función `BuscaTesoros` y mandar llamar las funciones del evaluador `Preguntar` y `Cavar` para completar tu tarea.

Internamente el evaluador llevará el registro de cuántos tesoros quedan. Tu programa no necesitará imprimir ni devolver nada: solo asegurarse de que hayas desenterrado los tesoros usando la función `Cavar`.

Implementación

```
void BuscaTesoros(int n, int m, int k);
```

Descripción: El evaluador buscará en tu código esta función y la llamará con los parámetros `n`, `m` y `k`. Tu implementación deberá utilizar las funciones `Preguntar` y `Cavar` para desenterrar todos los tesoros. En cada caso de prueba solo se llamará a esta función una vez.

Parámetros

- `n`: Filas de la cuadrícula.
- `m`: Columnas de la cuadrícula.
- `k`: El número de tesoros enterrados.

TODO COMPLETAR

Fuente: **OMI 2018**

www.omegaup.com/arena/problem/OMI2018-Tesoro

Problema E.2

⁴Este es un problema interactivo, si no conoces como trabajar con estos, ve la página: 113

Anexo

Problemas interactivos

Los problemas interactivos son unos problemas que se resuelven ligeramente diferente a los tradicionales.

En un problema tradicional, uno lee los datos, los procesa y luego obtenemos información de salida.

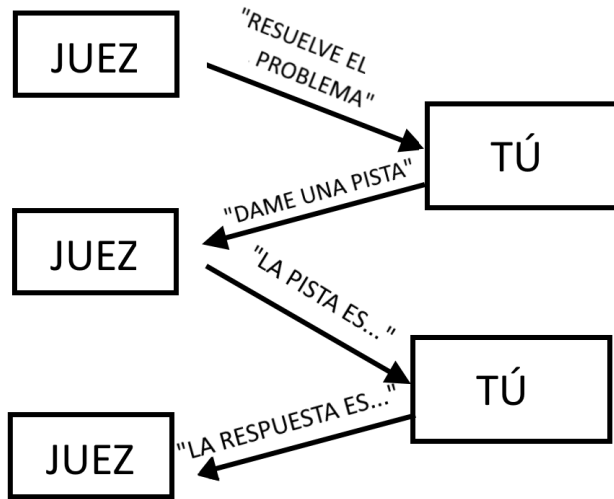


Pero esto no es como funciona en los problemas interactivos. En estos, puedes imaginar que hay dos programas, uno que es el juez y otro que es el tuyo.

Podemos imaginarlo como dos personas, una eres tú y la otra es el juez.

El juez te pedirá que hagas algo, como encontrar un número secreto. Y tu deberás hacer lo que te pide, pero también podrás pedirle favores al juez, ya sea que haga algo o que te responda una pregunta. Es decir, a diferencia de los problemas estándar, aquí puedes interactuar con el juez.

El flujo de información va en dos sentidos.



Como esto es implementado difiere un poco de juez a juez, pero en la OMI y la IOI se logra con el uso de funciones.

Tu tarea será implementar funciones que deben hacer lo que te pida el problema. Estas serán las que el juez utilizara para interactuar contigo.

Por ejemplo "resuelve el problema" podría verse como una función: `int resuelve(int N)` que se llamará al inicio para indicarte que resuelvas el problema, deberá retornar la solución.

Mientras que el problema te dirá que podrás llamar a funciones implementadas por el evaluador. Estas son las funciones con las cuales tú te comunicarás con el juez.

Por ejemplo: `int pista(int x)` es una función que te regresará algún tipo de pista que tu podrás usar para calcular la respuesta.

Veamos un ejemplo:

Ejemplo: Cluedo

El Dr. Negro ha sido asesinado. La detective Jill debe determinar el culpable, la ubicación y el arma del crimen. Existen seis posibles culpables, numerados del 1 al 6. Hay diez posibles ubicaciones, numeradas del 1 al 10. Existen 6 posibles armas homicidas, numeradas del 1 al 6. Para ilustrar, se muestran los nombres de los posibles culpables, ubicaciones y armas del crimen. Los nombres no son requeridos para resolver esta tarea.

Culpable	Ubicación	Arma Homicida
1. Profesor Ciruela 2. Señorita Escarlata 3. Coronel Mostaza 4. Señora Blanca 5. Reverendo verde 6. Señora Pavo Real	1. Salón 2. Comedor 3. Cocina 4. Conservatorio 5. Sala de Billar 6. Biblioteca 7. Sala de Baile 8. Vestíbulo 9. Estudio 10. Bodega	1. Daga 2. Soga 3. Revólver 4. Candelabro 5. Llave Inglesa 6. Tubería

Jill intenta repetidas veces adivinar la combinación correcta de culpable, ubicación y arma homicida. Cada intento de adivinar es llamado una teoría. Ella le pregunta a su asistente Jack que confirme o refute la teoría que tiene. Cuando Jack confirma una teoría, Jill ha resuelto el caso. Cuando Jack refuta una teoría, le presenta a Jill evidencia de que alguno de los elementos de la teoría es incorrecto.

Tu tarea es implementar un procedimiento `ResolverCaso()` que juegue el papel de Jill. El evaluador llamará a tu procedimiento `ResolverCaso()` varias veces, cada vez con un nuevo caso por resolver. `ResolverCaso()` puede llamar repetidas veces a la función `Teoria(c, u, a)`, que está implementada de manera interna en el evaluador. `c`, `u` y `a` son enteros denotando una combinación en particular de culpable, ubicación y arma homicida. `Teoria(c, u, a)` devolverá un entero, 0 si la teoría que presentes es correcta. Si la teoría es incorrecta, devolverá un valor 1, 2, o 3. El valor 1 indica que el culpable es incorrecto; 2 indica que la ubicación es incorrecta; 3 indica que el arma homicida es incorrecta. Si más de un elemento en la teoría es incorrecto, Jack elige uno al azar entre los que estén mal. Cuando una llamada a `Teoria(c, u, a)` devuelva 0, tu procedimiento `ResolverCaso()`

deberá retornar.

Implementación

Tu procedimiento: `void ResolverCaso()`

Descripción

Cada vez que Jill reciba un nuevo caso se llamará a este procedimiento, el cual deberás resolver. *Asegurate de inicializar cualquier variable que uses durante cada una de las llamadas a este procedimiento.*

Parámetros

- Este procedimiento no recibe parámetros.

Función del evaluador: `int Teoria(int c, int u, int a)`

Descripción

Llama a esta función para probar una teoría sobre el homicidio. La función te devolverá un valor entero. 0 cuando la teoría es correcta (recuerda terminar la ejecución cuando resuelvas el caso); 1 si el culpable es incorrecto; 2 si la ubicación es incorrecta; 3 si el arma homicida es incorrecta.

Parámetros

- c: La suposición del sospechoso.
- u: La suposición de la ubicación.
- a: La suposición del arma homicida.

Rutina de ejemplo

Como ejemplo, asume que la Señorita Escarlata asesinó al Dr. Negro en la cocina usando un candelabro. Cuando el procedimiento `ResolverCaso()` haga las siguientes teorías, los valores devueltos serán:

Funcion	Valor retornado	Explicación
ResolverCaso()	-	Jill tiene un nuevo caso por resolver.
Teoria(1, 1, 1)	1, 2 o 3	Todo es incorrecto
Teoria(3, 3, 3)	1 o 3	Solo la ubicación es correcta
Teoria(5, 3, 4)	1	Solo el culpable es incorrecto
Teoria(2, 3, 4)	0	Todo es correcto.

Límites

Se llamará a tu procedimiento ResolverCaso() un máximo de 360 veces.

Subtareas

Subtarea 1 [50 puntos]

Por cada llamada a ResolverCaso(), puedes llamar a Teoria() a lo más 360 veces.

Subtarea 2 [50 puntos]

Por cada llamada a ResolverCaso(), puedes llamar a Teoria() a lo más 20 veces.

Fuente: IOI 2010

www.omegaup.com/arena/problem/Cluedo

Compile y pruebe interactivos

En omegaup, el juez utilizado para la OMI hay dos formas de programar y trabajar con problemas interactivos.

La primera forma es descargar la plantilla en la parte de abajo del problema. Elige C++ y tu sistema operativo.

Se te descargará un archivo comprimido que deberás descomprimir en alguna carpeta de tu elección.

En esta carpeta verás que dentro hay varios archivos y folders, pero a nosotros solo nos interesan unos cuantos.

Principalmente, debemos ver un archivo `.cpp`: `cluedo.cpp`, este será el archivo en el cual deberás programar y enviar a omegaup.

Si tenemos CodeBlocks instalado, recomendamos utilizar este, para esto abres el archivo `.cbp`, una vez abierto esto verás a la izquierda los archivos de interés, `Sources/problem.cpp`.

Si no tenemos CodeBlocks, podemos trabajar de todas formas usando algún editor de texto y teniendo el compilador de C++ instalado, abriendo el archivo que tenga el nombre del problema + `.cpp`

Para `cluedo`, este archivo se llamará `cluedo.cpp` y al abrirlo verás:

```
#include "cluedo.h"

// Main
// int Teoria(int c, int u, int a)

void ResolverCaso() {
    // FIXME
}
```

La función `ResolverCaso` que dice `FIXME` es donde deberás escribir el código que resuelva el problema, para saber que debe hacer `ResolverCaso` lee la parte de implementación del problema.

El comentario que dice `//MAIN` enlista las funciones que existen a tu disposición implementadas por el evaluador. Estas funciones las tienes disponibles gracias al `#include"cluedo.h"`, es importante no borrar esa línea de código.

Ahora pasemos a resolver el problema:

La primera solución que ha de venir en mente es hacer una búsqueda completa, probando todas las combinaciones de asesino, ubicación y arma

hasta que encontremos la correcta. Como hay $6 \times 10 \times 6 = 360$ combinaciones, obtendríamos 50 puntos por esta idea.

Esto en código se vería:

```
#include "cluedo.h"

void ResolverCaso() {
    for (int c=1; c<=6; c++) {
        for (int u=1; u<=10; u++) {
            for (int a=1; a<=6; a++) {
                if (Teoria(c, u, a)==0) {
                    return;
                }
            }
        }
    }
}
```

Ahora, para probar tu código, lo que haces es ejecutarlo. Si estas en CodeBlocks, basta con compilar y ejecutar tu proyecto. Si no estas usando CodeBlocks, deberás ejecutar el script `compilar` y luego el script `test` desde la consola.

El programa probara tu código contra un caso de prueba y te dará información sobre como se desempeño. Si quieres cambiar el caso de ejemplo que se usa, deberás cambiar el archivo `examples/sample.in`.

Para entender como se prueba tu código también podrás consultar y modificar el archivo `Main.cpp`, aunque no es necesario es útil para añadir información extra de debug y entender el `sample.in`.

La solución a este problema se deja como reto al lector.

Colas

La cola es una estructura de datos bastante útil para muchas técnicas y problemas, en este libro se le da el principal uso para la BFS.

La idea es muy sencilla, queremos implementar algo que se comporte como la cola en la que uno se forma para comprar boletos.

En una fila pueden suceder tres cosas: alguien nuevo se forma, el de enfrente de la fila sale de la fila y el vendedor puede ver quien esta enfrente de la fila. Nosotros queremos que nuestra cola soporte esto eventos.

Entonces la cola es una estructura que puede hacer:

- Agregar algo al final de la cola.
- Quitar al que este enfrente de la cola.
- Ver que está enfrente de la cola.
- Saber cuantas cosas hay en la cola.

Nosotros podemos o implementar nuestra propia cola, o utilizar una que ya viene incluida en C++. En este libro usaremos la que viene en C++, pero en el libro de Estructura de Datos veremos como implementarla nosotros.

Usar queue

Para utilizar la cola, primero debemos incluir la librería `<queue>`. Una vez hecho esto, tendremos acceso a la cola de C++.

Para crear una cola, basta con declararla bajo un nombre y especificar que tipo de datos recibirá, para esto seguimos el formato `queue<tipo> nombre;`. Veamos un ejemplo de una cola de enteros.

```
| queue<int> colaDeEnteros;
```

A esta cola le podemos hacer las cuatro operaciones fundamentales y una extra:

- `.push(x)` Agregar x al final.
- `.pop()` Quitar al frente.
- `.front()` Ver el valor de enfrente.
- `.size()` Obtener cuantas cosas hay en la cola.
- `.empty()` Regresa verdadero si no hay elementos en la cola.

Veamos estas operaciones en uso

```
#include<iostream>
#include<queue>
using namespace std;
int main() {
    queue<int> cola;
    cout << "cola inicia vacia: " << cola.size() << "\n";
    cola.push(5);
    cola.push(3);
    cola.push(2);
    // La cola ahora tiene al {5, 3, 2} dentro
    cout << "\n";
    cout << "Tenemos " << cola.size() << " elementos\n";
    cout << "Enfrente esta: " << cola.front() << "\n";
    cola.pop();
    // La cola ahora tiene {3, 2}

    cout << "\n";
    cout << "Quedan " << cola.size() << " elementos\n";
    cout << "Enfrente esta: " << cola.front() << "\n";
    cola.push(5);
    cola.push(2);
    //La cola ahora tiene {3,2,5,2}
    cout << "\n";
    cout << "Imprimamos y vaciemos la cola:\n";
    while (cola.empty() == false) {
        cout << " " << cola.front() << "\n";
        cola.pop();
    }
}
```



```
    }  
    return 0;  
}
```

Este programa imprime:

cola inicia vacia: 0

Tenemos 3 elementos

Enfrente esta: 5

Quedan 2 elementos

Enfrente esta: 3

Imprimamos y vaciemos la cola:

3

2

5

2