**Búsqueda completa, lineal y binaria**

**Búsquedas**

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, un 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 de la lista uno que cumpla alguna propiedad especifica. Es decir, queremos encontrar una respuesta dentro de una lista de candidatos.

Entonces, búsqueda es simplemente encontrar una respuesta dentro de una lista de 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 tenemos un arreglo A de enteros distintos y este tiene N elementos en él. Nosotros querremos hacer un código que imprima la posición del arreglo que valga K. O si esta posición no existe, que imprima -1.

**Interpretación**Primero, entendamos que lo que el problema pide es buscar el índice del valor K dentro del arreglo A.

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

**Código**

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 candidatos es iguala a , entonces la complejidad es .

**Ejemplo 1.2**

Veamos otro problema para aprender búsqueda lineal.

**Descripción**

Carlos quiere armar una fiesta, y como le gusta ser un buen host compro 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 , indicando cuantos regalos compró Carlos.

**Salida**La cantidad de posibles números de invitados para la fiesta.

**Límites**

Antes de ver la solución, inténtalo resolver tu mismo.

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

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

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

**Código**

respuesta = 0;

for (int i =1; i <= N; i++) {

if (N%i==0) {

respuesta++;

}

}

cout << respuesta

**Problemas de practica**

Problema 1.1: Dado una lista de enteros, cuenta cuantas veces aparece el valor .   
Ejemplo:

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

Problema 1.3: Cuenta cuantos divisores positivos tiene , pero con límites:   
.

Problema 1.4: Encuentra el entero más grande que sea menor igual que y que la suma de sus dígitos sea menor o igual que .

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

Escalera de altura 3 que usa 6 ladrillos
Escalera de altura 5 que usa 15 ladrillos

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

Ejemplo

|  |  |
| --- | --- |
| K | Respuesta |
| 8 | 3 |
| 12 | 4 |
| 20 | 5 |

**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 simple, 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**

Karel tiene listones de distintas longitudes. Karel quiere hacer pulseras con estos listones, por lo que tomará cada uno de los listones y los cortará en segmentos del mismo tamaño entero sin que sobre nada de listón. ¿Cuántos diferentes tamaños de segmento se pueden elegir?

**Entrada**Un entero , indicando la cantidad de listones

En la siguiente línea, enteros indicando las longitudes de los listones. Llamemos a la longitud del listón .

**Salida**  
La cantidad de opciones para el tamaño de los segmentos.

**Limites**

**Solución**Encontremos con búsqueda lineal todos los tamaños de segmento que cumplen y contémoslo.

Primero veamos que los tamaños de listón deben estar entre y . Más concreto, entre y . 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.

Entonces ya hemos visto como se ve una búsqueda lineal y 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 listón más  
 pequeño. \*/

}

respuesta = 0;

for (int s =1; s <= minA; s++) {

if (es s es un tamaño de segmento válido) {

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 sin que sobre, ¿qué tiene que cumplir con relación al listón? Así es, que es, que divida a la longitud del listón. Y podemos ver que tiene que cumplir esto para todos los listones

Entonces, para ver que sea una opción válida, hay que ver que 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;

}

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: .

Como y . Nos queda que , lo cual corre en menos de un segundo.

**Conclusión**

Habrá 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.

**Problemas de practica**

Problema 1.6: Karel ha comprado una bicicleta eléctrica con la que planea completar un recorrido. El recorrido se puede ver como colinas en línea recta, cada una de altura . Karel comienza de la colina de 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 inicio hay una estación de recarga donde Karel puede recargar su bicicleta. Como nota, la batería tiene capacidad y jamás podrá almacena más energía que .

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

Ejemplo:

Limites:

Fuente: OMIS online 2022

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 . Tampoco 30 es capicúa.

Determina cuantos números son capicúas entre y

Ejemplo:

Problema 1.8: Dado y , encuentra cuantos números primos hay entre y , incluyendo y .

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

Ejemplo:  
Los primos contados para el ejemplo son 2,3,5 y 7

**Búsqueda completa**

La búsqueda completa 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, tiende a no ser una solución para los 100 puntos. Sin embargo, es muy útil conocerla ya que la fuerza bruta tiende a ser una subtarea y una forma de obtener puntos en un problema donde no se nos ocurra ideas mejores, también hay varios problemas que consisten en empezar de la fuerza bruta e ir mejorando la solución hasta que sea suficientemente buena.

También llega a ser útil para encontrar errores en una solución que tengamos, ya que podemos comparar resultados.

Probablemente esto te suene a búsqueda lineal, y esto es resultado de que búsqueda lineal es un tipo de búsqueda completa en la que revisamos secuencialmente un rango donde se puede encontrar la respuesta. Pero búsqueda completa incluye más tipos de iteraciones que solo revisar los valores de un ciclo for.

Comencemos viendo problemas en los que tengamos que buscar pares de elementos.

**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 solucionarlo. Para este tipo de problemas será útil conocer como hacer una búsqueda completa que revise todos los posibles pares de puntos.

Para ver esto en acción veamos un problema.

**Ejemplo 2.1**

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

Sin embargo, la ferretería vende cajas de tornillos, cada una con tornillos dentro.

Como Fernando tiene una obsesión con no desperdiciar, Fernando solo comprara las cajas de forma de que traigan juntas exactamente 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 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 de cada caja

**Salida**

Deberás imprimir “SI” si Fernando puede obtener K tornillos con sus reglas, o “NO” si es imposible.

**Ejemplo**

|  |  |
| --- | --- |
| Entrada | Salida |
| 5 6  3 1 8 5 | SI |
| 5 10  3 1 8 5 | NO |

**Límites**

**Solución**

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

Para determinar si existe dicha pareja, lo que haremos será buscar entre todas las parejas de cajas aquella que sume . 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 , tal que . Como queremos iterar por todos los posibles pares, primero iteraremos por todos los posibles valores de . Y para cada , iteraremos por todas las con las que 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 .

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 los que nos permiten iterar por toda pareja de elementos de un arreglo. Esta es una herramienta bastante útil para resolver muchos problemas y subtareas.

**Complejidad**

Bien, ahora hablemos de la complejidad de esta técnica de búsqueda completa. En concreto, para esta técnica la complejidad es .

La complejidad de esto es porque la cantidad de parejas con N elementos crece en .

Pero incluso si no conocemos como crecen las parejas, podemos ver que este ciclo para i =0, itera por valores de j; para , iteramos por valores de ; para , iteramos por valores de j. y así. De forma que hacemos iteraciones de . Entonces hacemos iteraciones.

Usando la formula se suma de gauss obtenemos que:

Entonces, la complejidad queda como

Entonces, iterar por todos los pares de un arreglo es una técnica de complejidad cuadrada, perfecta para limites hasta .

**Problemas de practica**

Problema 2.1: Definición: Inversión es la cantidad de parejas en un arreglo tal que y también .

Dado un arreglo de enteros, imprime cuantas inversiones tiene.

Ejemplo:

Problema 2.2: AQUÍ FALTA UN PROBLEMA 2.2

Problema 2.3: Fernando regreso otro día a la tienda de tornillos, y esta vez quiere darle una puntuación en una página de guías locales. Fernando juzga la calidad de la tienda en función de cuantas cantidades diferentes de tornillos puede comprar.

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

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

Problema 2.4: 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.  
De forma que los primeros números de la secuencia son:

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

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

Entrada:  
En la primera línea vendrá el valor de

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

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

Ejemplo:

|  |  |
| --- | --- |
| Entrada | Salida |
| 3  5  6  10 | SI NO SI |

**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.**

**Iterar por subconjuntos**

Ahora que ya sabemos de subconjuntos, aprendamos a revisarlos todos en una búsqueda completa.

Hay dos formas principales de iterar por un subconjunto, una recursiva y otra iterativa.

**Problemas de practica**