Búsqueda Binaria

Leonel Zeballos Andy Ortiz

18 de mayo de 2025

Universidad Mayor de San Simón

Taller de Programación Competitiva 2025

Contenidos

Búsqueda Lineal

Problema

Problema

Se te dan dos números n y k seguidos de un arreglo A de n enteros ordenados de forma creciente. Imprimir la posición del elemento k en el arreglo. Si el elemento no está en el arreglo imprimir -1.

Ejemplo

Entrada	Salida
9 8 0 1 3 5 7 8 10 10 11	5
2 3 1 2	-1

Solución

Podemos iterar en cada elemento de A hasta encontrar el valor k.
Una vez encontrado guardamos el índice y detenemos la búsqueda.

Solución

Podemos iterar en cada elemento de A hasta encontrar el valor k. Una vez encontrado guardamos el índice y detenemos la búsqueda.

```
int res = -1;
for (int i = 0; i < n; i++) {
    if (A[i] == k) {
        res = i;
        break;
    }
}
cout << res << '\n';
```

Solución

Podemos iterar en cada elemento de A hasta encontrar el valor k. Una vez encontrado guardamos el índice y detenemos la búsqueda.

```
int res = -1;
for (int i = 0; i < n; i++) {
    if (A[i] == k) {
        res = i;
        break;
    }
}
cout << res << '\n';
```

- **Complejidad:** $O(n)$

Observaciones

Puntos a considerar:

Observaciones

Puntos a considerar:

- ¿Cuál es el valor máximo de n? ¿Es suficiente $O(n)$?

Observaciones

Puntos a considerar:

- ¿Cuál es el valor máximo de n? ¿Es suficiente $O(n)$?
  - Para un $n \leq 10^8$ ✓

Observaciones

Puntos a considerar:

- ¿Cuál es el valor máximo de n? ¿Es suficiente $O(n)$?
  - Para un $n \leq 10^8$ ✓
  - Para un $n \leq 10^{18}$ ✗

Observaciones

Puntos a considerar:

- ¿Cuál es el valor máximo de n? ¿Es suficiente $O(n)$?
  - Para un $n \leq 10^8$ ✓
  - Para un $n \leq 10^{18}$ ✗
- ¿El orden de los elementos me sirve de algo?

Observaciones

Puntos a considerar:

- ¿Cuál es el valor máximo de n? ¿Es suficiente $O(n)$?
  - Para un $n \leq 10^8$ ✓
  - Para un $n \leq 10^{18}$ ✗
- ¿El orden de los elementos me sirve de algo?
  - Si, podemos aprovechar la propiedad de orden . . . ¿Cómo?

Búsqueda Binaria

Observación

Dado que los elementos en A están ordenados, es decir, se cumple que $a_i \leq a_j$ para todo $i < j$. Podemos notar que:

Observación

Dado que los elementos en A están ordenados, es decir, se cumple que $a_i \leq a_j$ para todo $i < j$. Podemos notar que:

- Si a_i es **mayor** que k entonces todos los elementos que le siguen son también mayores.

$$\text{Si } a_i > k \text{ entonces } a_x > k \ \forall x : i \leq x < n$$

Observación

Dado que los elementos en A están ordenados, es decir, se cumple que $a_i \leq a_j$ para todo $i < j$. Podemos notar que:

- Si a_i es **mayor** que k entonces todos los elementos que le siguen son también mayores.

$$\text{Si } a_i > k \text{ entonces } a_x > k \ \forall x : i \leq x < n$$

- Si a_i es **menor** que k entonces todos los elementos que le preceden son también menores.

$$\text{Si } a_i < k \text{ entonces } a_x < k \ \forall x : 0 \leq x \leq i$$

Búsqueda Binaria

Podemos descartar/podar el espacio de búsqueda gracias a la propiedad de orden.

Búsqueda Binaria

Búsqueda Binaria

Podemos descartar/podar el espacio de búsqueda gracias a la propiedad de orden.

Búsqueda Binaria

- Mantenemos dos índices $l < r$ tal que nuestro espacio de búsqueda es: $a_{l+1}, a_{l+2}, \ldots, a_{r-2}, a_{r-1}$

Importante

Notar que a_l y a_r no forman parte de nuestro espacio de búsqueda.

Búsqueda Binaria

- Probamos si el elemento k se encuentra al **medio** del espacio de búsqueda, es decir, si $a_m = k$. Se puede calcular el índice central m entre l y r de las siguientes dos formas:

$$m = \left\lfloor \frac{l + r}{2} \right\rfloor \tag{1}$$

$$m = l + \left\lfloor \frac{r - l}{2} \right\rfloor \tag{2}$$

  De las formas mostradas se sugiere utilizar la 2, esto debido a problemas con el overflow que puede causar $l + r$ si estos son enteros muy grandes.

Búsqueda Binaria

- Si $a_m \neq k$ entonces comprobamos dos casos:

Búsqueda Binaria

- Si $a_m \neq k$ entonces comprobamos dos casos:
  - Si $a_m > k$ entonces descartamos todos los elementos que le siguen a a_m (incluyendo a a_m). Esto es sencillo, solo movemos r a donde estaba m, ahora $r = m$.

Búsqueda Binaria

- Si $a_m \neq k$ entonces comprobamos dos casos:
  - Si $a_m > k$ entonces descartamos todos los elementos que le siguen a a_m (incluyendo a a_m). Esto es sencillo, solo movemos r a donde estaba m, ahora $r = m$.
  - Si $a_m < k$ entonces descartamos todos los elementos que preceden a a_m (incluyendo a a_m). Esto es sencillo, solo movemos l a donde estaba m, ahora $l = m$.

Búsqueda Binaria

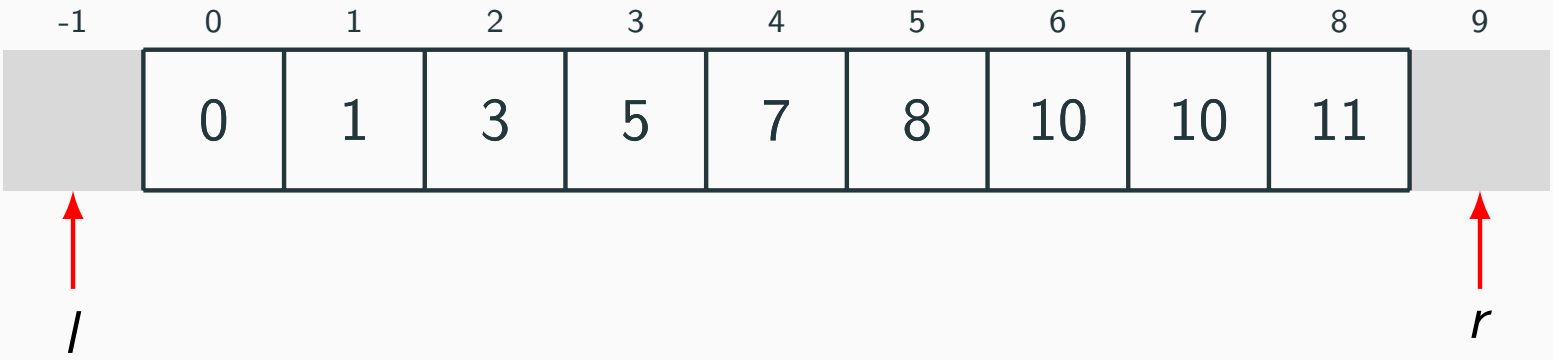
- Si $a_m \neq k$ entonces comprobamos dos casos:
  - Si $a_m > k$ entonces descartamos todos los elementos que le siguen a a_m (incluyendo a a_m). Esto es sencillo, solo movemos r a donde estaba m, ahora $r = m$.
  - Si $a_m < k$ entonces descartamos todos los elementos que preceden a a_m (incluyendo a a_m). Esto es sencillo, solo movemos l a donde estaba m, ahora $l = m$.
- Repetimos este proceso hasta que nos encontremos con el elemento k o en el último de los casos, hasta que nos quedemos sin un espacio de búsqueda. Sabemos que tenemos aún espacio de búsqueda si $r - l > 1$.

Ejemplo

Veamos un ejemplo para el primer caso del problema inicial. Observe que como no queremos considerar a a_l y a_r como parte del espacio de búsqueda, definimos inicialmente $l = -1$ y $r = n$ (índices no válidos en un arreglo de n elementos).

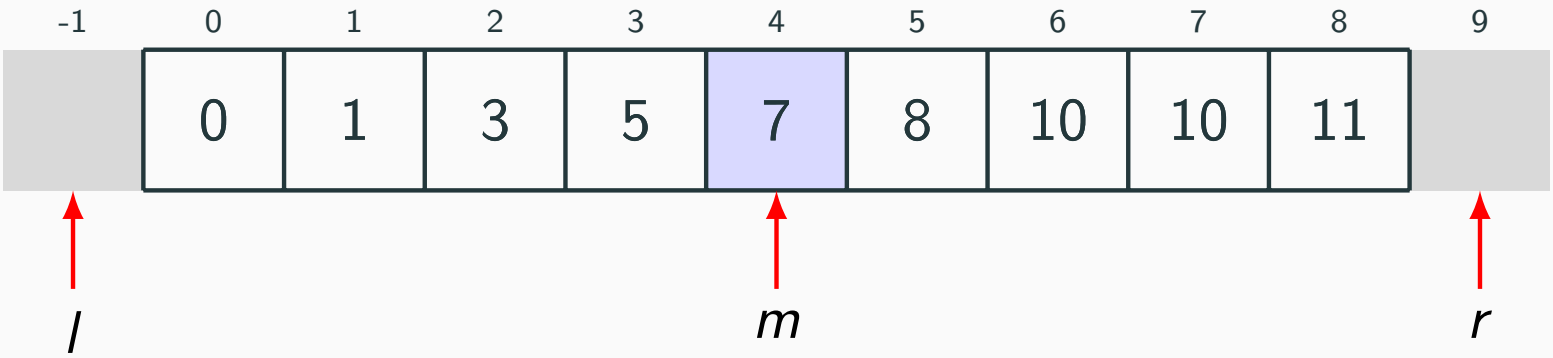
Ejemplo

$$k = 8$$



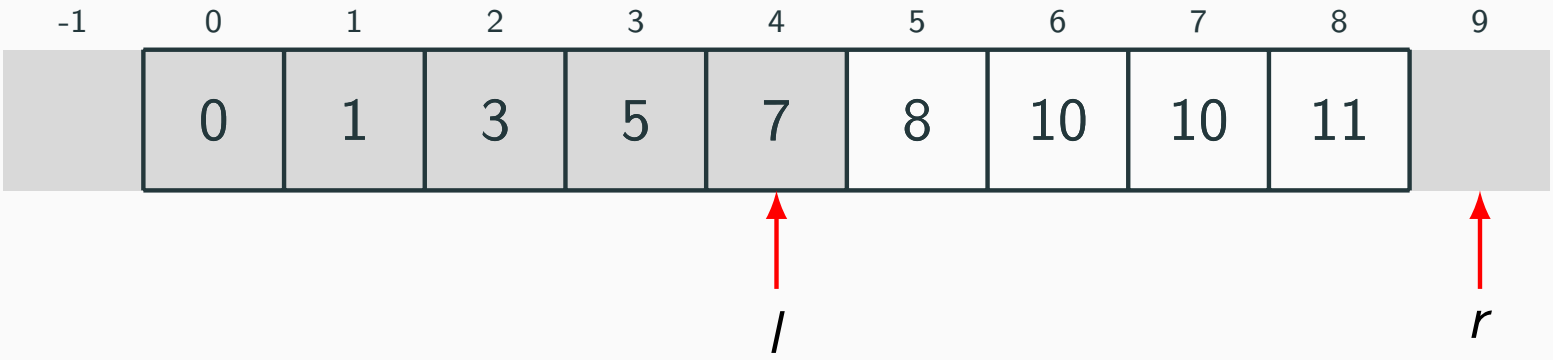
Ejemplo

$$k = 8$$



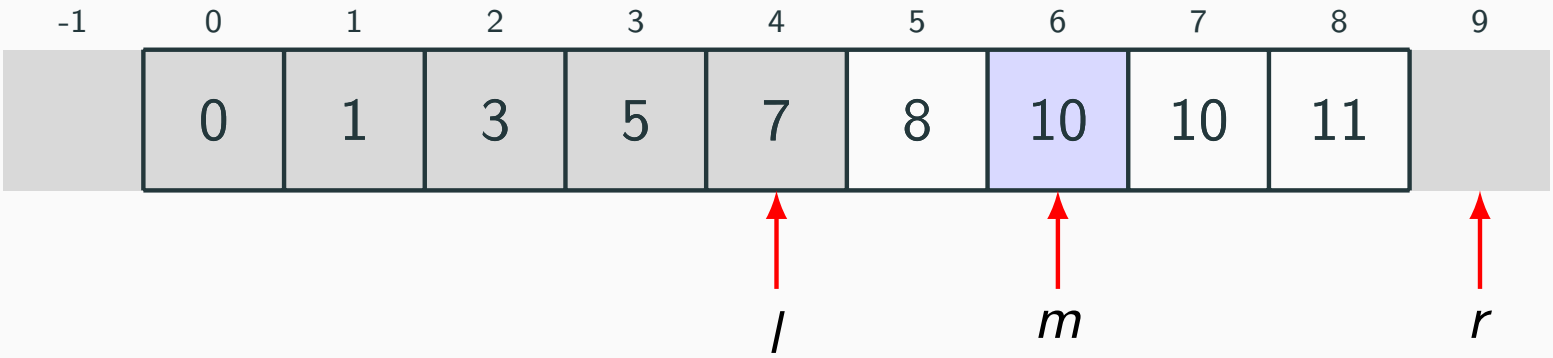
Ejemplo

$$k = 8$$



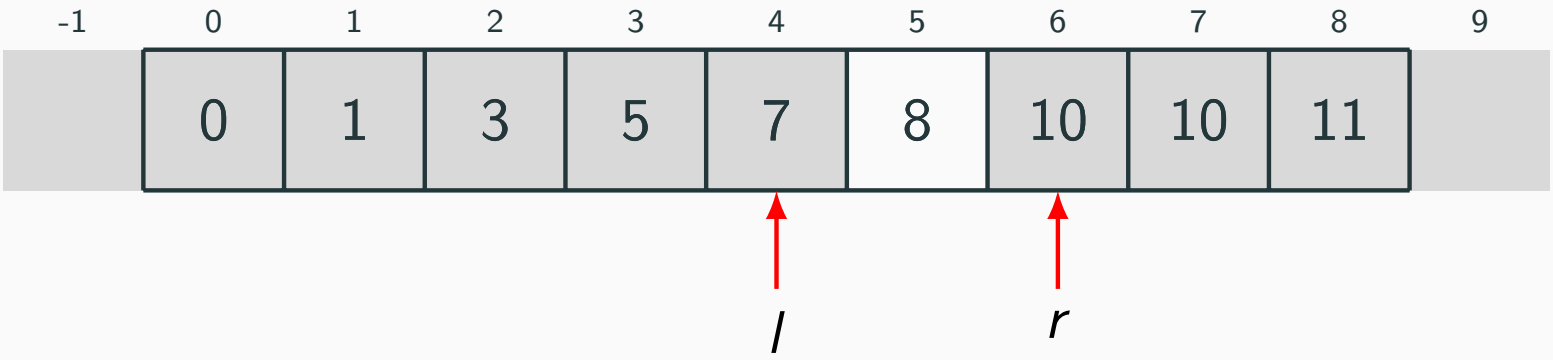
Ejemplo

$$k = 8$$



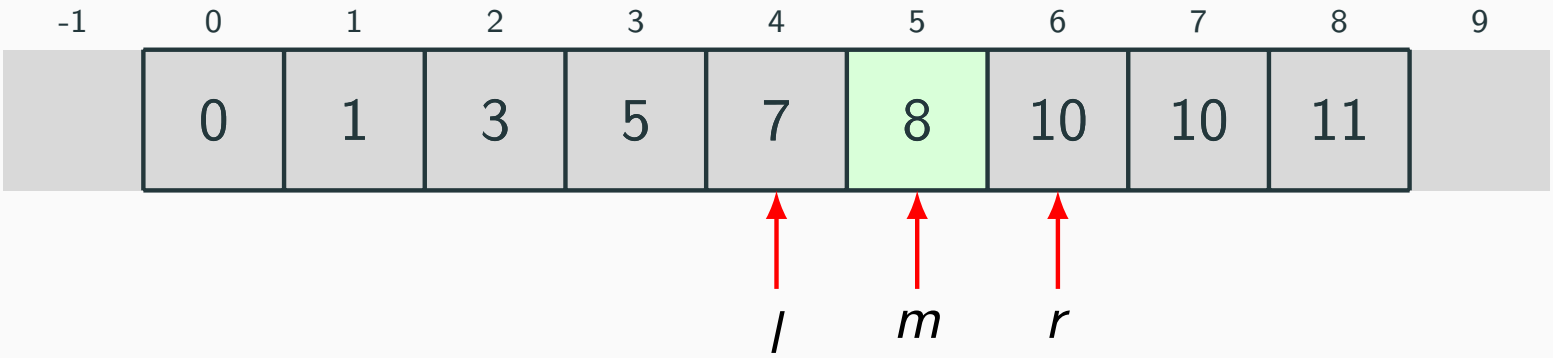
Ejemplo

$$k = 8$$



Ejemplo

$$k = 8$$



Implementación

```
int res = -1;
int l = -1, r = n;
while (r - l > 1) {
    int m = l + (r - l) / 2;
    if (a[m] == k) {
        res = m;
        break;
    }
    if (a[m] > k) {
        r = m;
    } else {
        l = m;
    }
}
```

Implementación

Existe más de una forma para implementar la búsqueda binaria, sin embargo, es muy susceptible a pequeños errores en la implementación que conducen a comportamientos no esperados. La implementación brindada anteriormente es una forma limpia y correcta para la búsqueda binaria.

Más formas de implementar en: Competitive Programmer's Handbook (página 31)

Complejidad

Análisis

Analicemos como cambia el tamaño de nuestro espacio de búsqueda en cada iteración de la búsqueda.

Análisis

Analicemos como cambia el tamaño de nuestro espacio de búsqueda en cada iteración de la búsqueda.

- Inicialmente tenemos n elementos en los que buscar nuestra respuesta.

Análisis

Analicemos como cambia el tamaño de nuestro espacio de búsqueda en cada iteración de la búsqueda.

- Inicialmente tenemos n elementos en los que buscar nuestra respuesta.
- Desplazamos alguno de los índices l o r a la mitad del intervalo, quedándonos con solo $\frac{n}{2}$ elementos.

Análisis

Analicemos como cambia el tamaño de nuestro espacio de búsqueda en cada iteración de la búsqueda.

- Inicialmente tenemos n elementos en los que buscar nuestra respuesta.
- Desplazamos alguno de los índices l o r a la mitad del intervalo, quedándonos con solo $\frac{n}{2}$ elementos.
- Volvemos a desplazar algún índice a la mitad del intervalo, quedándonos con solo $\frac{n}{4}$ elementos.

Análisis

Analicemos como cambia el tamaño de nuestro espacio de búsqueda en cada iteración de la búsqueda.

- Inicialmente tenemos n elementos en los que buscar nuestra respuesta.
- Desplazamos alguno de los índices l o r a la mitad del intervalo, quedándonos con solo $\frac{n}{2}$ elementos.
- Volvemos a desplazar algún índice a la mitad del intervalo, quedándonos con solo $\frac{n}{4}$ elementos.
- ...
- Finalmente llegamos a quedarnos con aproximadamente un elemento. La idea era reducir lo más posible nuestro espacio de búsqueda.

Análisis

En la primera iteración nos quedamos con $\frac{n}{2}$ elementos, en la segunda con $\frac{n}{4}$, en la tercera con $\frac{n}{8}$, y así sucesivamente hasta que $n \approx 1$. Por lo tanto, si h es la cantidad de veces que debemos dividir entre dos el espacio de búsqueda hasta quedarnos con aproximadamente un elemento (la respuesta en el mejor caso), podemos hallar h de la siguiente forma:

Análisis

En la primera iteración nos quedamos con $\frac{n}{2}$ elementos, en la segunda con $\frac{n}{4}$, en la tercera con $\frac{n}{8}$, y así sucesivamente hasta que $n \approx 1$. Por lo tanto, si h es la cantidad de veces que debemos dividir entre dos el espacio de búsqueda hasta quedarnos con aproximadamente un elemento (la respuesta en el mejor caso), podemos hallar h de la siguiente forma:

$$\frac{n}{2^h} \approx 1$$

$$n \approx 2^h$$

$$log_2 n \approx h$$

Análisis

En la primera iteración nos quedamos con $\frac{n}{2}$ elementos, en la segunda con $\frac{n}{4}$, en la tercera con $\frac{n}{8}$, y así sucesivamente hasta que $n \approx 1$. Por lo tanto, si h es la cantidad de veces que debemos dividir entre dos el espacio de búsqueda hasta quedarnos con aproximadamente un elemento (la respuesta en el mejor caso), podemos hallar h de la siguiente forma:

$$\frac{n}{2^h} \approx 1$$

$$n \approx 2^h$$

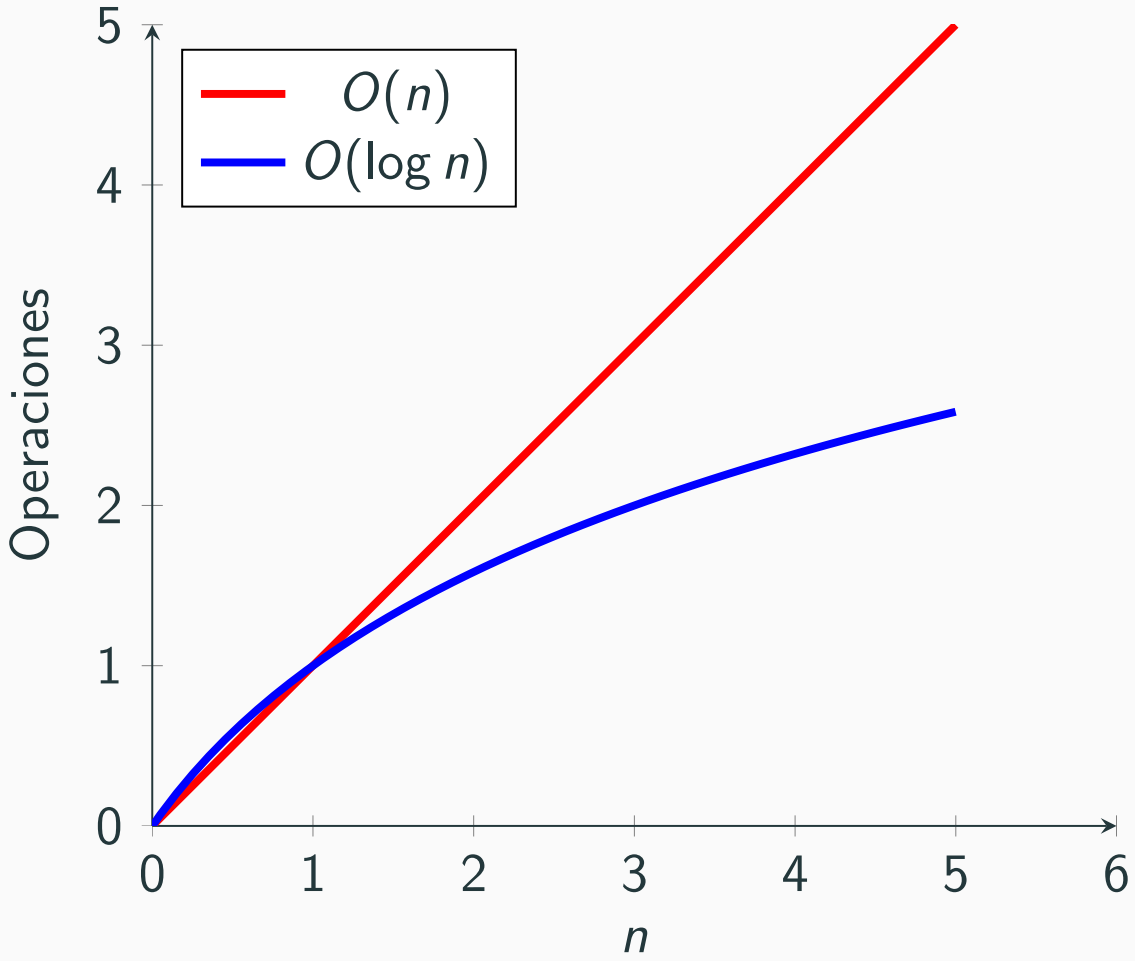
$$log_2 n \approx h$$

Entonces necesitamos realizar aproximadamente $log_2 n$ o $logn$ iteraciones/operaciones como máximo hasta agotar el espacio de búsqueda.

Complejidad

La complejidad entonces es: $O(logn)$

Comparación



Comparación

Tabla 1: Comparación en la cantidad de operaciones para $O(n)$ y $O(logn)$ con distintos tamaños de n.

n	$O(n)$	$O(logn)$
1	1	1
5	5	3
50	50	6
10^3	10^3	10
10^8	10^8	27
10^{18}	10^{18}	60

Ejercicios

Problema

Ejercicio 1

Se te da un arreglo de n números enteros ordenados de forma creciente y un número entero x, encuentra la posición del primer elemento del arreglo que sea mayor o igual a x. Si el elemento no existe en el arreglo imprimir -1.

Ejemplo

Entrada	Salida
6 5 1 3 5 7 9 11	2
6 12 1 3 5 7 9 11	-1

Solución

```
int l = -1, r = a.size();
int ans = -1;
while( r - l > 1){
    int mid =  l + (r - l) / 2;
    if(a[mid] >= x){
        ans = mid;
        r = mid;
    }else{
        l = mid;
    }
}
cout << ans << '\n';
```

Solución

```
int l = -1, r = a.size();
int ans = -1;
while( r - l > 1){
    int mid =  l + (r - l) / 2;
    if(a[mid] >= x){
        ans = mid;
        r = mid;
    }else{
        l = mid;
    }
}
cout << ans << '\n';
```

- **Complejidad:** $O(logn)$

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.
- Si no existe tal valor, se imprime -1.

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.
- Si no existe tal valor, se imprime `-1`.
- La variable `ans` guarda la mejor posición candidata mientras se busca.

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.
- Si no existe tal valor, se imprime `-1`.
- La variable `ans` guarda la mejor posición candidata mientras se busca.
- En cada iteración, si $a[\texttt{mid}] \geq x$, intentamos buscar un índice menor (lado izquierdo).

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.
- Si no existe tal valor, se imprime `-1`.
- La variable `ans` guarda la mejor posición candidata mientras se busca.
- En cada iteración, si $a[\texttt{mid}] \geq x$, intentamos buscar un índice menor (lado izquierdo).
- El límite izquierdo `l = -1` y derecho `r = n` permiten cubrir todos los índices posibles correctamente.

Observaciones

- El arreglo está ordenado crecientemente, lo cual permite usar búsqueda binaria.
- Se busca el primer índice tal que $a[i] \geq x$.
- Si no existe tal valor, se imprime `-1`.
- La variable `ans` guarda la mejor posición candidata mientras se busca.
- En cada iteración, si $a[\texttt{mid}] \geq x$, intentamos buscar un índice menor (lado izquierdo).
- El límite izquierdo `l = -1` y derecho `r = n` permiten cubrir todos los índices posibles correctamente.
- La condición `r - l > 1` garantiza que el algoritmo termina.

Problema

Ejercicio 2

Se te da un arreglo binario a (solo contiene 1's y 0's) de tamaño n, ordenados de forma no creciente(todos los 1s primero y luego los 0's). Encuentra la posición de la última apararición de 1. Si no hay 1's imprime -1.

Ejemplo

Entrada	Salida
7 1 1 1 1 0 0 0	3
5 0 0 0 0 0	-1

Solución

```
int l = -1, r = a.size();
while (r - l > 1) {
    int mid = l + (r - l) / 2;
    if (a[mid] == 1) {
        l = mid;
    } else {
        r = mid;
    }
}
cout << l << '\n';
```

Solución

```
int l = -1, r = a.size();
while (r - l > 1) {
    int mid = l + (r - l) / 2;
    if (a[mid] == 1) {
        l = mid;
    } else {
        r = mid;
    }
}
cout << l << '\n';
```

- **Complejidad:** $O(logn)$

Observaciones

- El arreglo está ordenado no crecientemente: los 1s están antes que los 0s.

Observaciones

- El arreglo está ordenado no crecientemente: los 1s están antes que los 0s.
- El objetivo es encontrar el último índice i tal que $a[i] = 1$.

Observaciones

- El arreglo está ordenado no crecientemente: los 1s están antes que los 0s.
- El objetivo es encontrar el último índice i tal que $a[i] = 1$.
- Si $a[\texttt{mid}] = 1$, significa que la última aparición de 1 está en *mid* o más a la derecha, por eso se actualiza `l = mid`.

Observaciones

- El arreglo está ordenado no crecientemente: los 1s están antes que los 0s.
- El objetivo es encontrar el último índice i tal que $a[i] = 1$.
- Si $a[\texttt{mid}] = 1$, significa que la última aparición de 1 está en *mid* o más a la derecha, por eso se actualiza `l = mid`.
- Si $a[\texttt{mid}] = 0$, entonces el último 1 debe estar a la izquierda de *mid*, así que se actualiza `r = mid`.

Observaciones

- El arreglo está ordenado no crecientemente: los 1s están antes que los 0s.
- El objetivo es encontrar el último índice i tal que $a[i] = 1$.
- Si $a[\texttt{mid}] = 1$, significa que la última aparición de 1 está en *mid* o más a la derecha, por eso se actualiza `l = mid`.
- Si $a[\texttt{mid}] = 0$, entonces el último 1 debe estar a la izquierda de *mid*, así que se actualiza `r = mid`.
- Al finalizar el bucle, *l* será la posición del último 1. Si no hay 1s, `l = -1`.

Problema

Ejercicio 3

Tienes n paquetes con pesos $a_1, a_2, \ldots, a_n$, y k camiones. Cada camión debe recibir una secuencia contigua de paquetes. Todos los paquetes deben ser transportados y cada camión debe llevar al menos uno.

Encuentra la capacidad mínima que deben tener los camiones para que se pueda hacer la distribución sin exceder esa capacidad.

Ejemplo

Entrada	Salida
5 3 1 2 3 4 5	6
4 2 7 2 5 10	14

Solución

```
int l = 1, r = INF, ans = r;
    while (r - l > 1) {
        int mid = l + (r - l) / 2;
        if (distribucion(libros, k, mid)) {
            ans = mid;
            r = mid;
        } else {
            l = mid;
        }
    }
cout << ans << '\n';
```

Solución

```
bool distribucion(vector<int>& libros, int k, int cap){
    int usado = 1, suma = 0;
    for(int x: libros){
        if( x > cap ) return false;
        if( suma + x <= cap ){
            suma += x;
        }else{
            usado++;
            suma = x;
        }
    }
    return usado <= k;
}
```

Solución

```
bool distribucion(vector<int>& libros, int k, int cap){
    int usado = 1, suma = 0;
    for(int x: libros){
        if( x > cap ) return false;
        if( suma + x <= cap ){
            suma += x;
        }else{
            usado++;
            suma = x;
        }
    }
    return usado <= k;
}
```

- **Complejidad:** $O(n \log s)$

Observaciones

- Utilizamos búsqueda binaria sobre la capacidad, con el rango de búsqueda de ‘left $= 1$‘ (capacidad mínima) y ‘right $=$ INF‘ (capacidad máxima).

Observaciones

- Utilizamos búsqueda binaria sobre la capacidad, con el rango de búsqueda de ‘left $= 1$‘ (capacidad mínima) y ‘right $=$ INF‘ (capacidad máxima).
- Para cada posible capacidad ‘mid‘, usamos la función `distribucion` para verificar si podemos repartir los paquetes entre los k camiones sin exceder la capacidad.

Observaciones

- Utilizamos búsqueda binaria sobre la capacidad, con el rango de búsqueda de ‘left $= 1$‘ (capacidad mínima) y ‘right $=$ INF‘ (capacidad máxima).
- Para cada posible capacidad ‘mid‘, usamos la función `distribucion` para verificar si podemos repartir los paquetes entre los k camiones sin exceder la capacidad.
- La función `distribucion` simula la asignación de paquetes a camiones: si al agregar un paquete se supera la capacidad, se inicia un nuevo camión.

Observaciones

- Utilizamos búsqueda binaria sobre la capacidad, con el rango de búsqueda de ‘left $= 1$‘ (capacidad mínima) y ‘right $=$ INF‘ (capacidad máxima).
- Para cada posible capacidad ‘mid‘, usamos la función `distribucion` para verificar si podemos repartir los paquetes entre los k camiones sin exceder la capacidad.
- La función `distribucion` simula la asignación de paquetes a camiones: si al agregar un paquete se supera la capacidad, se inicia un nuevo camión.
- Si podemos distribuir los paquetes con la capacidad ‘mid‘, entonces intentamos una capacidad menor (esto es el objetivo de la búsqueda binaria).

Observaciones

- Utilizamos búsqueda binaria sobre la capacidad, con el rango de búsqueda de ‘left $= 1$‘ (capacidad mínima) y ‘right $=$ INF‘ (capacidad máxima).
- Para cada posible capacidad ‘mid‘, usamos la función `distribucion` para verificar si podemos repartir los paquetes entre los k camiones sin exceder la capacidad.
- La función `distribucion` simula la asignación de paquetes a camiones: si al agregar un paquete se supera la capacidad, se inicia un nuevo camión.
- Si podemos distribuir los paquetes con la capacidad ‘mid‘, entonces intentamos una capacidad menor (esto es el objetivo de la búsqueda binaria).
- Si no podemos distribuir los paquetes, necesitamos aumentar la capacidad.

Problema

Ejercicio 4

Tienes n máquinas en una fábrica. La máquina i puede fabricar un producto cada t_i segundos. Todas las máquinas pueden trabajar simultáneamente desde el principio. ¿Cuál es el menor tiempo necesario para producir al menos k productos?

Ejemplo

Entrada	Salida
3 7 3 2 5	8
4 16 1 1 1 1	4

Solución

```
int l = 0, r = INF + 1;
while(r - l > 1){
        int mid = l + (r - l) / 2;
        if(sePuedeGenerar(a, p, mid, n)){
            r = mid;
        }else{
            l = mid;
        }
    }
cout << r << endl;
```

Solución

```
bool sePuedeGenerar(vi& a, int p, int t, int n){
    int cont = 0;
    for(int i = 0; i < n; i++){
        cont += t/a[i];
        if(cont >= p) return true;
    }
    return cont >= p;
}
```

Solución

```
bool sePuedeGenerar(vi& a, int p, int t, int n){
    int cont = 0;
    for(int i = 0; i < n; i++){
        cont += t/a[i];
        if(cont >= p) return true;
    }
    return cont >= p;
}
```

- **Complejidad:** $O(n \log s)$

Observaciones

- Utilizamos búsqueda binaria sobre el tiempo, con el rango de búsqueda desde `left = 0` (tiempo mínimo posible) hasta `right = INF + 1` (tiempo suficientemente grande).

Observaciones

- Utilizamos búsqueda binaria sobre el tiempo, con el rango de búsqueda desde `left = 0` (tiempo mínimo posible) hasta `right = INF + 1` (tiempo suficientemente grande).
- Para cada tiempo candidato `mid`, usamos una función que calcula cuántos productos se pueden fabricar en ese tiempo si todas las máquinas trabajan en paralelo.

Observaciones

- Utilizamos búsqueda binaria sobre el tiempo, con el rango de búsqueda desde `left = 0` (tiempo mínimo posible) hasta `right = INF + 1` (tiempo suficientemente grande).
- Para cada tiempo candidato `mid`, usamos una función que calcula cuántos productos se pueden fabricar en ese tiempo si todas las máquinas trabajan en paralelo.
- La función suma para cada máquina $\left\lfloor \frac{\texttt{mid}}{t_i} \right\rfloor$, que representa cuántos productos fabrica la máquina i en `mid` unidades de tiempo.

Observaciones

- Utilizamos búsqueda binaria sobre el tiempo, con el rango de búsqueda desde `left = 0` (tiempo mínimo posible) hasta `right = INF + 1` (tiempo suficientemente grande).
- Para cada tiempo candidato `mid`, usamos una función que calcula cuántos productos se pueden fabricar en ese tiempo si todas las máquinas trabajan en paralelo.
- La función suma para cada máquina $\left\lfloor \frac{\texttt{mid}}{t_i} \right\rfloor$, que representa cuántos productos fabrica la máquina i en `mid` unidades de tiempo.
- Si se pueden fabricar al menos k productos en el tiempo `mid`, entonces intentamos encontrar una mejor respuesta (menor tiempo), ajustando el límite superior de la búsqueda binaria.

Observaciones

- Utilizamos búsqueda binaria sobre el tiempo, con el rango de búsqueda desde `left = 0` (tiempo mínimo posible) hasta `right = INF + 1` (tiempo suficientemente grande).
- Para cada tiempo candidato `mid`, usamos una función que calcula cuántos productos se pueden fabricar en ese tiempo si todas las máquinas trabajan en paralelo.
- La función suma para cada máquina $\left\lfloor \frac{\texttt{mid}}{t_i} \right\rfloor$, que representa cuántos productos fabrica la máquina i en `mid` unidades de tiempo.
- Si se pueden fabricar al menos k productos en el tiempo `mid`, entonces intentamos encontrar una mejor respuesta (menor tiempo), ajustando el límite superior de la búsqueda binaria.
- Si no se puede cumplir la meta en `mid`, significa que necesitamos más tiempo, por lo tanto aumentamos el límite inferior.

Problema

Ejercicio 5

Vas a hornear una cantidad de galletas. Hay n ingredientes. Para hornear una galleta, necesitas exactamente a_i unidades del ingrediente i.

Actualmente tienes b_i unidades del ingrediente i en tu cocina. Además, tienes k unidades de polvo mágico. Cada unidad de polvo mágico puede reemplazar una unidad de cualquier ingrediente que te falte.

¿Cuál es la cantidad máxima de galletas que puedes hornear utilizando tus ingredientes y el polvo mágico?

Ejemplo

Ejemplo

Entrada	Salida
3 1 2 1 4 11 3 6	4
4 3 4 3 5 6 11 12 14 20	3

Solución

```
int l = 0, r = INF + 1;
    while(r - l > 1){
        int mid = l + (r - l) / 2;
        if(!cocinar(a, b, k, mid, n)){
            r = mid;
        }else{
            l = mid;
        }
    }
    cout << l << endl;
```

Solución

```
bool cocinar(vi& a, vi& b, int k, int mid, int n){
    for(int i = 0; i < n; i++){
        int x = mid * a[i], y = b[i];
        if(x > y){
            y += k;
            if(x <= y) k = y - x;
            else return false;
        }
    }
    return true;
}
```

Solución

```
bool cocinar(vi& a, vi& b, int k, int mid, int n){
    for(int i = 0; i < n; i++){
        int x = mid * a[i], y = b[i];
        if(x > y){
            y += k;
            if(x <= y) k = y - x;
            else return false;
        }
    }
    return true;
}
```

- **Complejidad:** $O(n \log s)$

Observaciones

- Utilizamos búsqueda binaria sobre la cantidad de galletas que se pueden hornear, con el rango de búsqueda desde `l = 0` hasta `r = INF + 1`.

Observaciones

- Utilizamos búsqueda binaria sobre la cantidad de galletas que se pueden hornear, con el rango de búsqueda desde `l = 0` hasta `r = INF + 1`.
- Para cada cantidad tentativa `mid`, usamos una función que verifica si es posible hacer `mid` galletas con los ingredientes disponibles y el polvo mágico.

Observaciones

- Utilizamos búsqueda binaria sobre la cantidad de galletas que se pueden hornear, con el rango de búsqueda desde `l = 0` hasta `r = INF + 1`.
- Para cada cantidad tentativa `mid`, usamos una función que verifica si es posible hacer `mid` galletas con los ingredientes disponibles y el polvo mágico.
- La función calcula cuántas unidades faltan de cada ingrediente y suma el total de polvo mágico necesario para cubrir ese déficit.

Observaciones

- Utilizamos búsqueda binaria sobre la cantidad de galletas que se pueden hornear, con el rango de búsqueda desde `l = 0` hasta `r = INF + 1`.
- Para cada cantidad tentativa `mid`, usamos una función que verifica si es posible hacer `mid` galletas con los ingredientes disponibles y el polvo mágico.
- La función calcula cuántas unidades faltan de cada ingrediente y suma el total de polvo mágico necesario para cubrir ese déficit.
- Si el polvo mágico necesario es menor o igual a k, entonces es posible hornear esa cantidad de galletas y se intenta una mayor cantidad.

Observaciones

- Utilizamos búsqueda binaria sobre la cantidad de galletas que se pueden hornear, con el rango de búsqueda desde `l = 0` hasta `r = INF + 1`.
- Para cada cantidad tentativa `mid`, usamos una función que verifica si es posible hacer `mid` galletas con los ingredientes disponibles y el polvo mágico.
- La función calcula cuántas unidades faltan de cada ingrediente y suma el total de polvo mágico necesario para cubrir ese déficit.
- Si el polvo mágico necesario es menor o igual a k, entonces es posible hornear esa cantidad de galletas y se intenta una mayor cantidad.
- Si el polvo necesario supera k, se intenta una menor cantidad de galletas.