BÚSQUEDA BINARIA SOBRE FUNCIONES

Leonardo Tovíaz
“Tobillos”

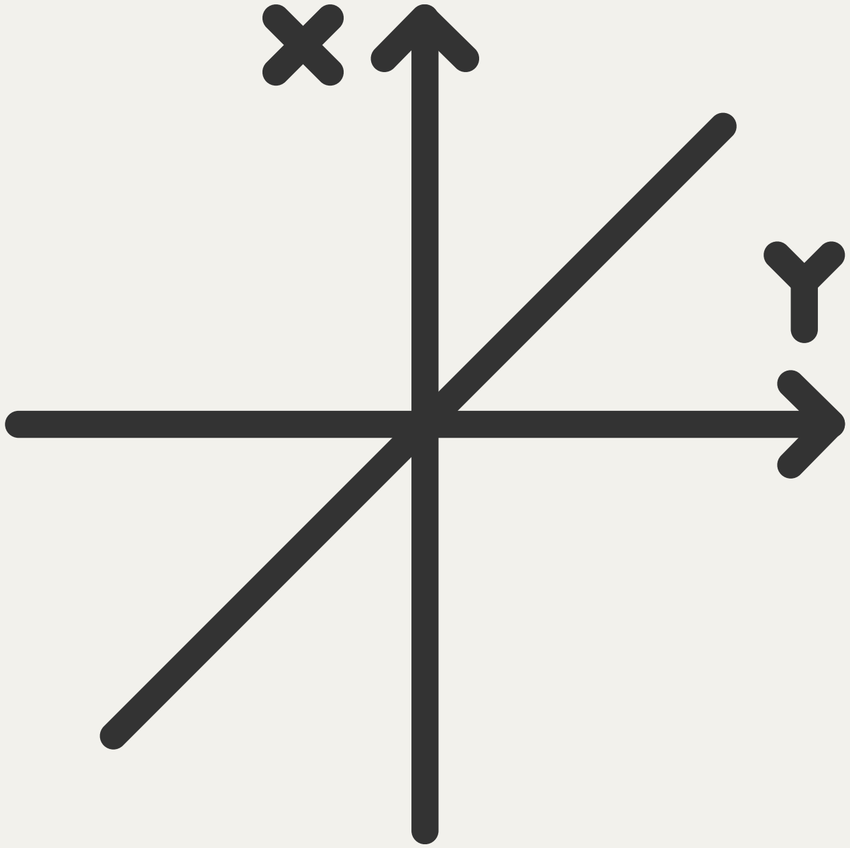
Club de programación
CSC

INTRODUCCIÓN

Primero veamos rápidamente la búsqueda binaria “normal” sobre un arreglo, de donde viene?

- Dado un arreglo [a0, a1...an-1], buscar si existe un número t.

```
for (int i = 0; i < a.length; i++) {
  int x = a[i];
  if (x == t) {
    print(“El número sí esta!”);
    break;
  }
}
```



DE O(N) A O(LOG N)

Esta versión lineal O(n) funciona perfecto pero se puede hacer mucho más rápido si se cumple una pequeña condición...

Que el arreglo esté ordenado, dentro de poco veremos un poco el porqué.

INTUICIÓN O IDEA

1. Imáginemos que tenemos un arreglo [1, 2, 3, 4, 5, 6] y que buscamos el valor 5.
2. Pensemos en que nuestro número podría estar a la mera mitad (escogamos mentalmente 3).
3. Notemos que 3 es menor a 5, entonces no tiene sentido buscar el 5 ahí o más a la izquierda (esto si nuestro arreglo esta ordenado).

Entonces “iniciamos” de nuevo la busqueda pero nuestro arreglo ahora “será” [4, 5, 6].. ahora que escojamos el número de enmedio de nuevo llegaremos al 5.

Notar que con búsqueda lineal hubieramos tenido que revisar 5 elementos (del 1 al 5) pero con esta búsqueda binaria solo tuvimos que revisar 2 (3 y 5).
Esto escala demasiado a como va creciendo el tamaño del arreglo...

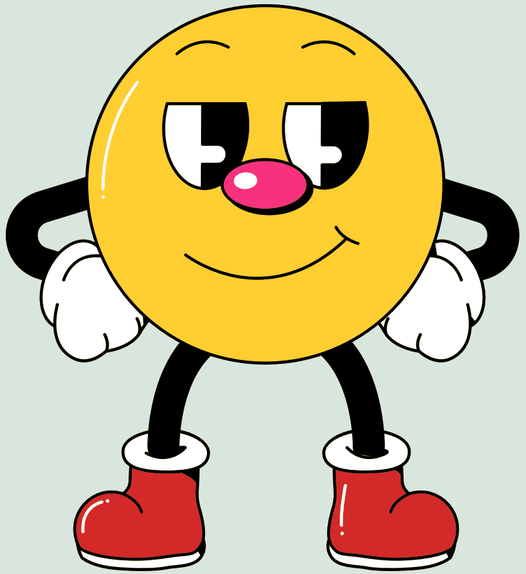
IMPLEMENTACIÓN

```cpp
int binsearch(vector<int>& nums, int target) {
  int l = 0; [redacted]
  int r = nums.size() - 1; [redacted]
  int res = -1;
  while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] == target) {
      return m;
    }
    if (nums[m] < target) { // buscar mas arriba
      l = m + 1;
    } else { // buscar mas abajo
      r = m - 1;
    }
  }
  return res;
}
```

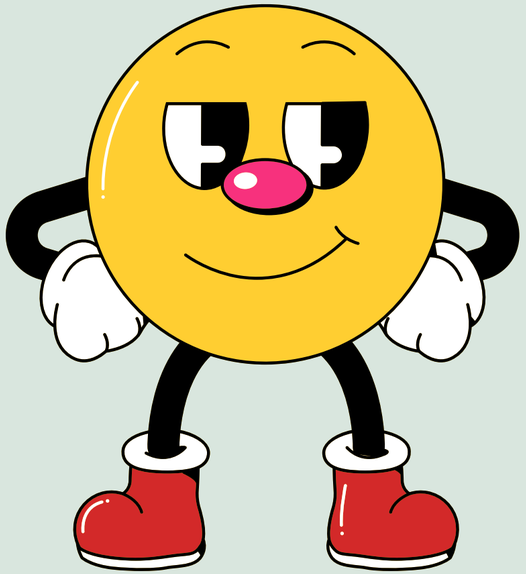
La primera vez que vemos este método se nos puede hacer demasiado, pero estas ideas pueden ayudar:

while → intervalo activo de busqueda
l → el menor indice donde puede estar t
r → el mayor indice donde puede estar t

Pero guarden los celulares por ahora, no le tomen foto a esto que no es tan importante.



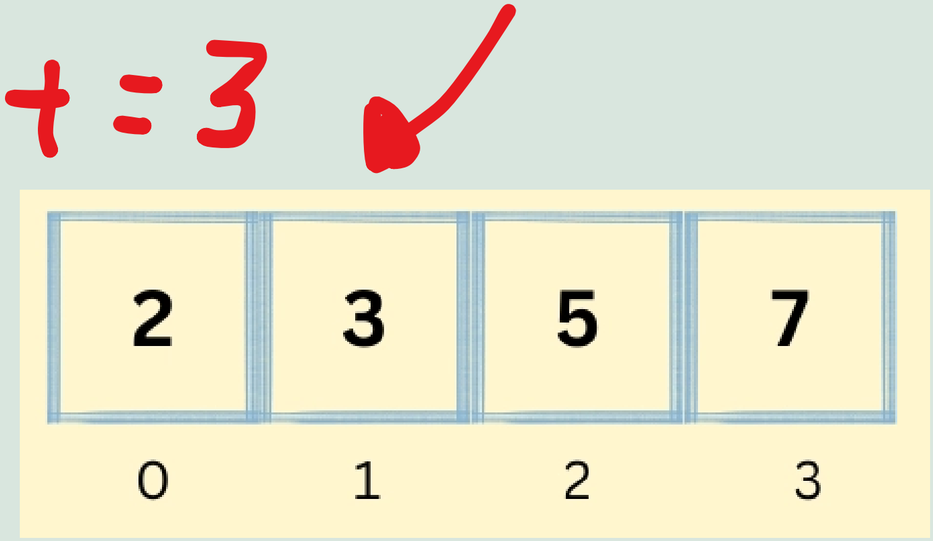
Bueno, pues la búsqueda binaria sobre un arreglo esta bien chida pero en los problemas de programación competitiva nunca nos van a pedir implementar una porque pues que chiste tiene nomas copiar y pegar (o aprendérselo de memoria).



De aquí, surgen 2 “modificaciones” a la búsqueda binaria sobre un arreglo que la harán muy útil para resolver problemas (y en una de esas nos servirá en alguna app millonaria que programemos).

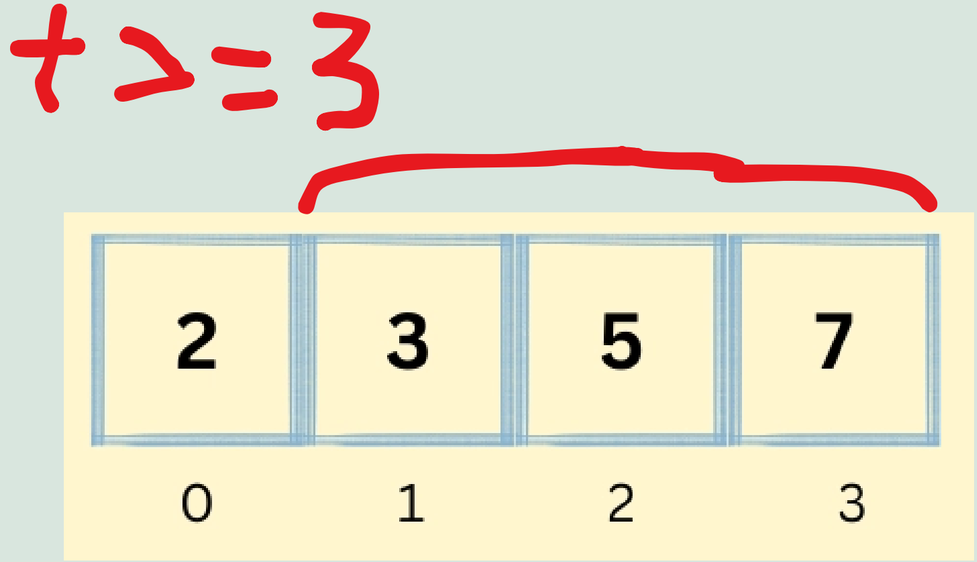
1. LOWER BOUND

Nuestra búsqueda binaria se basa en el operador = para encontrar el objetivo, esto implica que solo hay un resultado válido a lo mucho..



Qué pasaría si cambiamos este operador por un >=?
Cambian nuestros resultados válidos, es decir, ahora nuestra búsqueda se detendría cuando encuentre 3, 5 o 7 (en el caso de este arreglo).

Digamos que ahora reformulamos un “existe t en a?” a “existe un x >= t?”



1. LOWER BOUND

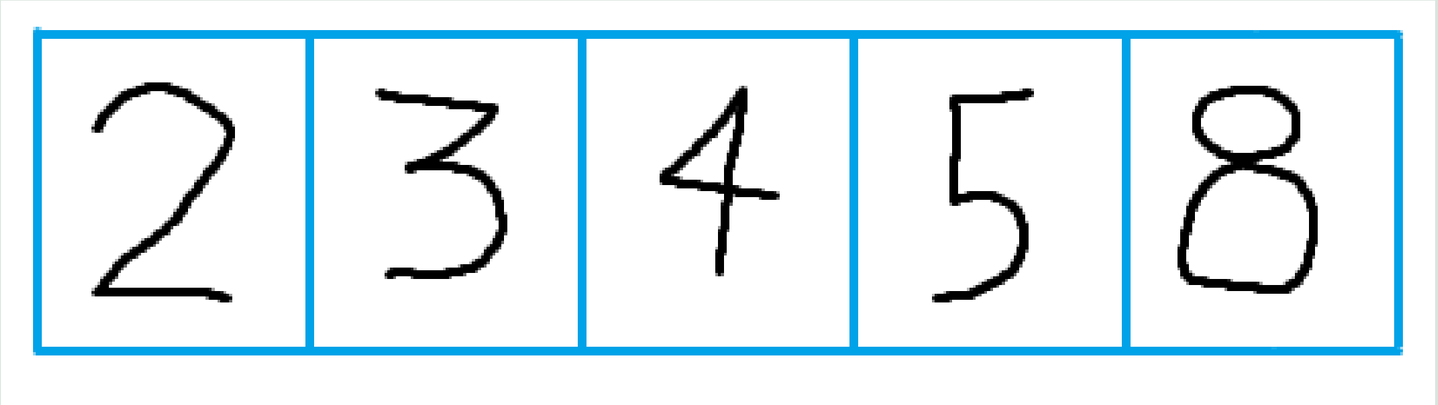
Y ahora, notemos que la implementación “estándar” de búsqueda binaria regresa ~el primer resultado válido encontrado~, lo cual tiene sentido si buscamos un número en específico..

Pero si tenemos varios resultados válidos nuestra búsqueda binaria puede estar tentada a regresar el primero, veamos:

```cpp
int binsearch(vector<int>& nums, int target) {
  int l = 0; // [illegible]
  int r = nums.size() - 1; // [illegible]
  int res = -1;
  while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] == target) {
      return m;
    }
    if (nums[m] < target) { // buscar mas arriba
      l = m + 1;
    } else { // buscar mas abajo
      r = m - 1;
    }
  }
  return res;
}
```

1. LOWER BOUND

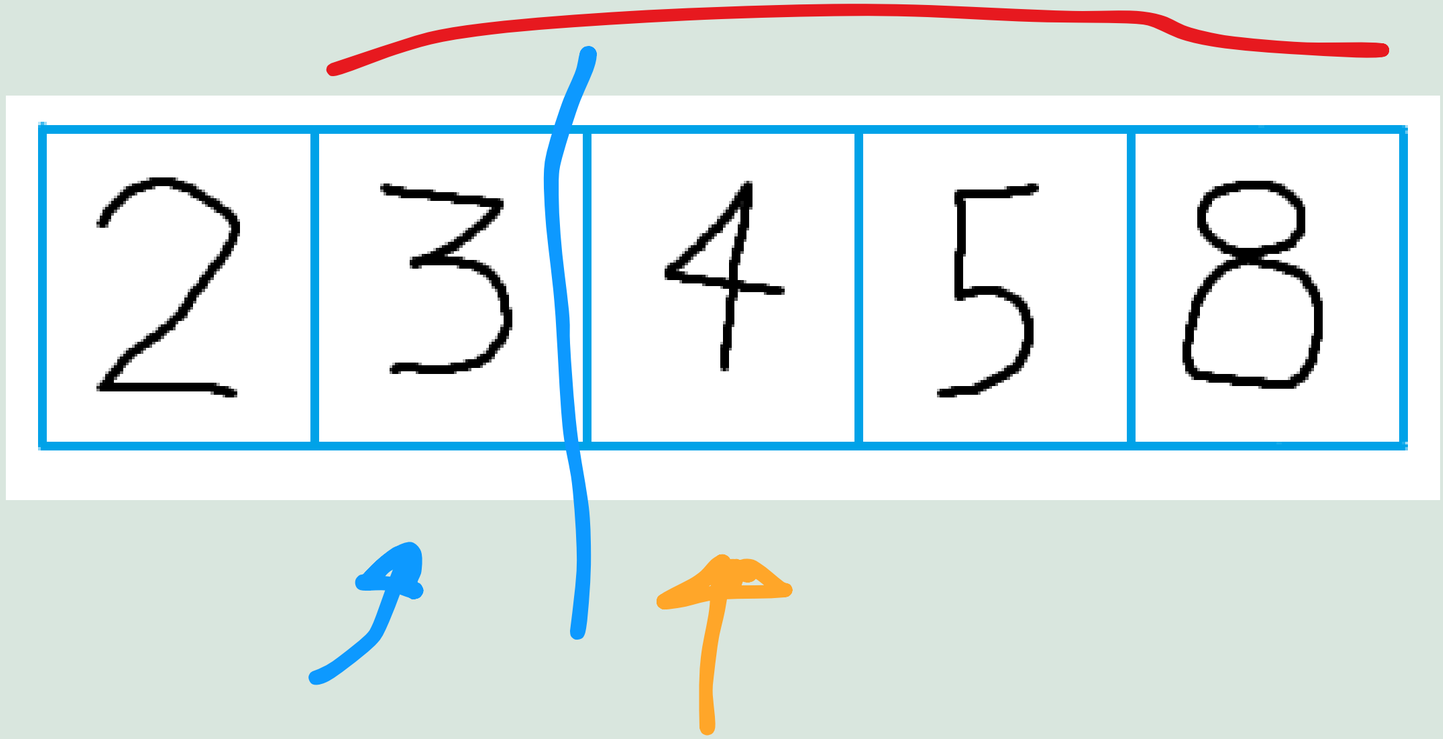
- ¿Cuáles son los resultados válidos p/t >= 3?
- ¿Cuál es el número que regresaría la función en el primer paso?
- Si buscamos el elemento más pequeño que cumple la condición, ¿ya lo encontramos solo con la primer revisión?



1. LOWER BOUND

- ¿Cuáles son los resultados válidos p/t >= 3?
- ¿Cuál es el número que regresaría la función en el primer paso?
- Si buscamos el elemento más pequeño que cumple la condición, ¿ya lo encontramos solo con la primer revisión?

NO



1. LOWER BOUND

```cpp
int binsearch(vector<int>& nums, int target) {
  int l = 0; // [redacted]
  int r = nums.size() - 1; // [redacted]
  int res = -1;
  while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] == target) {
      return m;
    }
    if (nums[m] < target) { // buscar mas arriba
      l = m + 1;
    } else { // buscar mas abajo
      r = m - 1;
    }
  }
  return res;
}
```

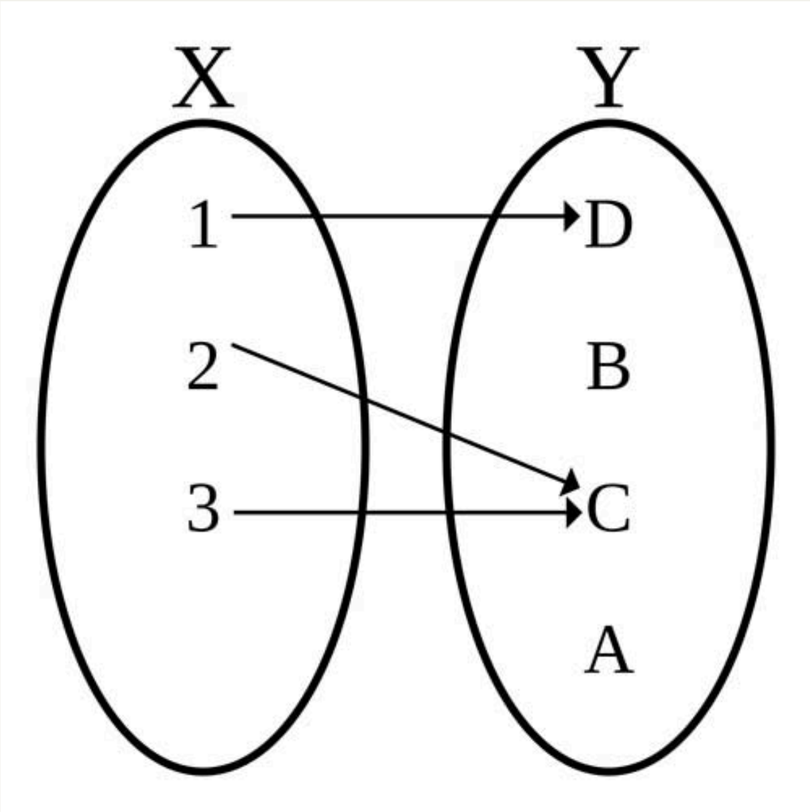
Versión “normal”

```cpp
int binsearch(vector<int>& nums, int target) {
  int l = 0; // [redacted]
  int r = nums.size() - 1; // [redacted]
  int res = -1;
  while (l <= r) {
    int m = l + (r - l) / 2;
    if (nums[m] >= target) {
      // se encontró un candidato,
      // buscar uno menor aún
      res = m;
      r = m - 1;
    } else {
      // el num no es igual o mayor a t,
      // quizá arriba haya
      l = m + 1;
    }
  }
  return res;
}
```

Versión lower_bound

Ya estamos más cerca de tener una búsqueda bínaria funcional, el pequeño problemita es que ningún problema nos va a dar un arreglo para luego pedirle el lower bound (porque quedaríamos igual, solo copiar y pegar).

Primero recordemos unos conceptos matemáticos.



CONCEPTOS

Función

Una “regla” que relaciona cada elemento de un conjunto A con un único elemento de un conjunto B.

*Varios elementos de A pueden estar relacionados a un mismo elemento en B.

- La regla que asigna a cada número su mitad.
  f(x) = x/2

- La regla que asigna a cada persona su CURP.
  f(persona) = CURP

- La función que asigna a cada número de la secuencia fibonacci el numero 1.
  f(n) = 1

CONCEPTOS (ANALOGÍA MÁQUINA)

Dominio	Conjunto de valores para los que está definida una función, en el ejemplo de la CURP, los números no forman parte de este.
Codominio	Conjunto de valores que teóricamente pueden “salir” de una función, en el primer ejemplo, cualquier número puede salir (porque los números son infinitos).
Imagen	Conjunto ”real” de valores que salen al ingresar a la función los elementos del dominio, en el primer ejemplo, la imagen cambiaría dependiendo del dominio.

CONCEPTOS (DEFINICIÓN DE REGLA)

Dominio	Valores del conjunto A.
Codominio	Valores del conjunto B (aunque no se usen todos).
Imagen	Subconjunto de B (valores que sí se usan).

..

Para la función mostrada..

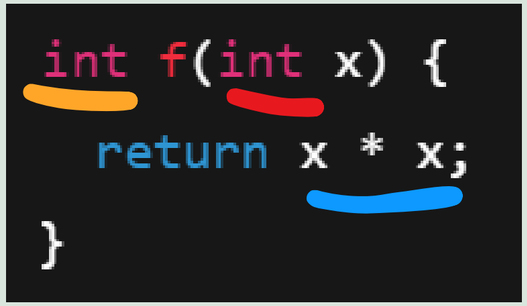
- ¿Cuál es su dominio?
- ¿Cuál es su codominio?
- ¿Cuál es su imagen?

```
int f(int x) {
  return x * x;
}
```

..

Para la función mostrada..

- ¿Cuál es su dominio?
- ¿Cuál es su codominio?
- ¿Cuál es su imagen?



- Todos los números enteros (la función acepta int).
- Todos los números enteros (al multiplicar dos números obtenemos otro número).
- Números positivos cuadrados (al multiplicar un numero positivo o negativo por sí mismo se cancela signo negativo si hay y se obtiene su cuadrado).

..

Para la función mostrada..

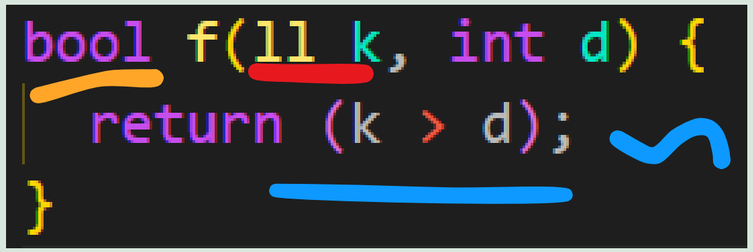
- ¿Cuál es su dominio?
- ¿Cuál es su codominio?
- ¿Cuál es su imagen?

```
bool f(ll k, int d) {
  return (k > d);
}
```

..

Para la función mostrada..

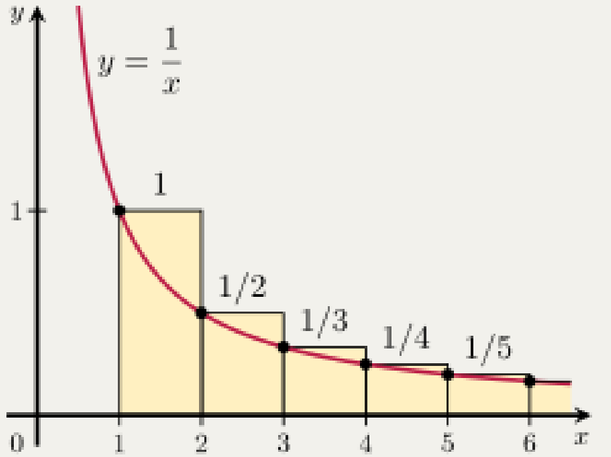
- ¿Cuál es su dominio?
- ¿Cuál es su codominio?
- ¿Cuál es su imagen?

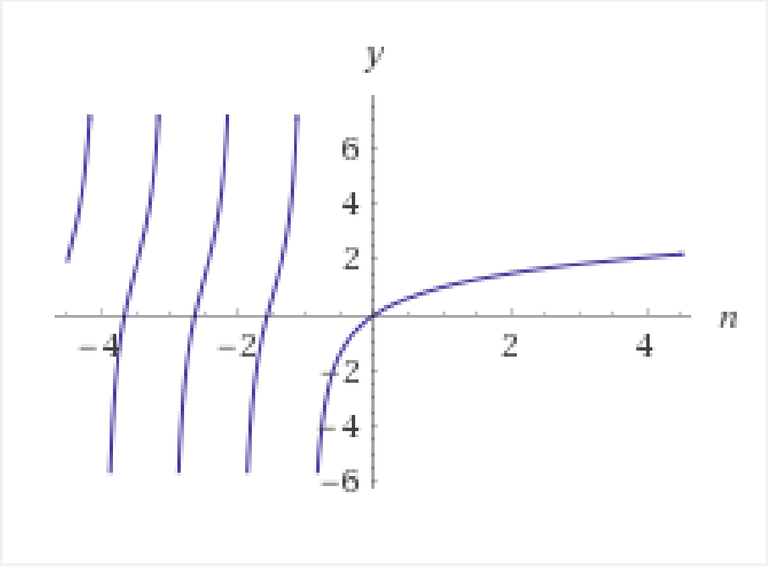


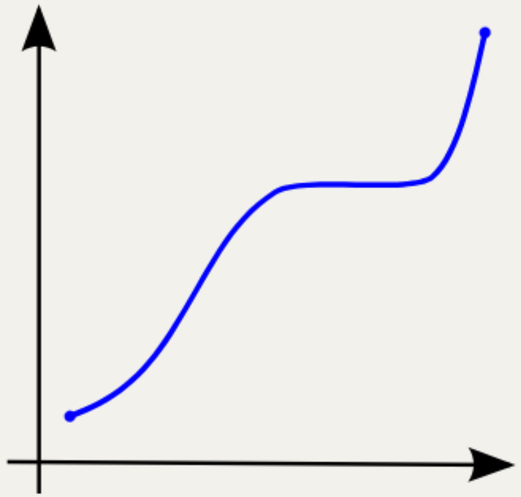
- Todos los números enteros (la función acepta ll; long long).
- Todos los valores booleanos (solo true y false).
- Depende del dominio, puede ser true y false o solo true o solo false (si siempre que evaluemos esta funcion usamos un k mayor a d, la imagen será true).

CONCEPTOS

Gráfica de una función	Es una línea que muestra los pares relacionados por la función (puntos del plano cartesiano). También nos muestra su “comportamiento”.







2. EVALUANDO FUNCIÓN

“La búsqueda binaria funciona en cualquier cosa que sea una función monótona”.

Una función monótona es una función cuya imagen tiene un comportamiento siempre ascendente o siempre descendente, aunque un comportamiento estático es válido también.

2. EVALUANDO FUNCIÓN

“La búsqueda binaria funciona en cualquier cosa que sea una función monótona”.

Una función monótona es una función cuya imagen tiene un comportamiento siempre ascendente o siempre descendente, aunque un comportamiento estático es válido también.

Antes de continuar, pensemos en que los arreglos ordenados son monótonos, por eso han funcionado las búsqueda binarias hasta el momento:

En un arreglo el dominio son los índices, y la imagen son los elementos dentro del arreglo.
“La función que asigna a cada índice un número”.

2. EVALUANDO FUNCIÓN

En los problemas de búsqueda binaria es muy raro que nos den un arreglo, entonces cual será nuestro dominio y codominio/imagen?

- Usaremos como dominio/entradas números que nosotros definiremos.
- Y como codominio/salidas solo tendremos true y false.

El objetivo es encontrar el primer número/número más pequeño que haga que una función regrese true.

¿Cuál función? También la definiremos nosotros, veamos un ejemplo.

2. EVALUANDO FUNCIÓN

Ejemplo:

Hay una fábrica con n máquinas y cada una produce una pieza cada ai segundos, ¿cuál es el mínimo tiempo para producir k piezas?

1. Nos piden un mínimo, solo con esto el problema ya es fuerte candidato a ser resuelto con binsearch.
2. Hay varios puntos de tiempo donde las k piezas ya estarán producidas (o incluso más), quizá en 30 minutos (t=30) hay 4 piezas, y quizá en 20 minutos (t=20) hay 4 piezas también, entonces deducimos que existe monotonicidad (varios t pueden llevar al mismo valor en la imagen).
3. Asumimos que una máquina tarda almenos 1 minuto en producir una pieza y máximo 1,000 minutos (estas restricciones nos las dá el problema).

2. EVALUANDO FUNCIÓN

Ejemplo:

Hay una fábrica con n máquinas y cada una produce una pieza cada ai segundos, ¿cuál es el mínimo tiempo para producir k piezas?

4. En el mejor caso, en el primer minuto se crearán las k piezas (quizá tenemos muchas máquinas muy eficientes y una demanda muy poca)... L=1
5. En el peor caso, nuestras máquinas se tardan muchísimo por su poca eficiencia y tenemos mucha demanda (este peor caso lo estimamos de acuerdo a las restricciones del problema; aquí podría ser: máximo tiempo de produccion de una pieza*k)... R=x*k

*Las restricciones podrían decirnos que ai=0, lo que significa que tenemos una máquina que no produce nada por minuto, lo que significa que habrá algún caso donde no se pueda satisfacer la demanda debido a que todas nuestras máquinas estarán descompuestas.

2. EVALUANDO FUNCIÓN

$$f(t) = \begin{cases} \text{true}, & \text{si en } t \text{ segundos se pueden producir al menos } k \text{ objetos} \\ \text{false}, & \text{si en } t \text{ segundos no se pueden producir } k \text{ objetos} \end{cases}$$

6. Defino una función de 2 partes como la de arriba.
7. “Convierto” la función a matemáticas y/o código, en este caso:

$$f(t) = \begin{cases} \text{true}, & \text{si } \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \geq k \\ \text{false}, & \text{si } \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor < k \end{cases}$$

..Sí, quizá parezca un conjuro demoniaco con tanto símbolo matemático pero solo es un ciclo...

2. EVALUANDO FUNCIÓN

1

$$f(t) = \begin{cases} \text{true}, & \text{si en } t \text{ segundos se pueden producir al menos } k \text{ objetos} \\ \text{false}, & \text{si en } t \text{ segundos no se pueden producir } k \text{ objetos} \end{cases}$$

2 Opcional.

$$f(t) = \begin{cases} \text{true}, & \text{si } \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \geq k \\ \text{false}, & \text{si } \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor < k \end{cases}$$

3

```cpp
bool f(ll t, vector<int> &maquinas, ll k) {
  ll total = 0;
  for (int i = 0; i < maquinas.size(); i++) {
    int p = maquinas[i];
    // cantidad de objetos producidos por la máquina p en t segundos,
    // podemos comprobarlo a mano
    total += t / p;
    // pequeña optimización lazy: si ya alcanzamos k, no necesitamos seguir sumando
    if (total >= k) return true;
  }
  return false;
}
```

2. EVALUANDO FUNCIÓN (SOLUCIÓN)

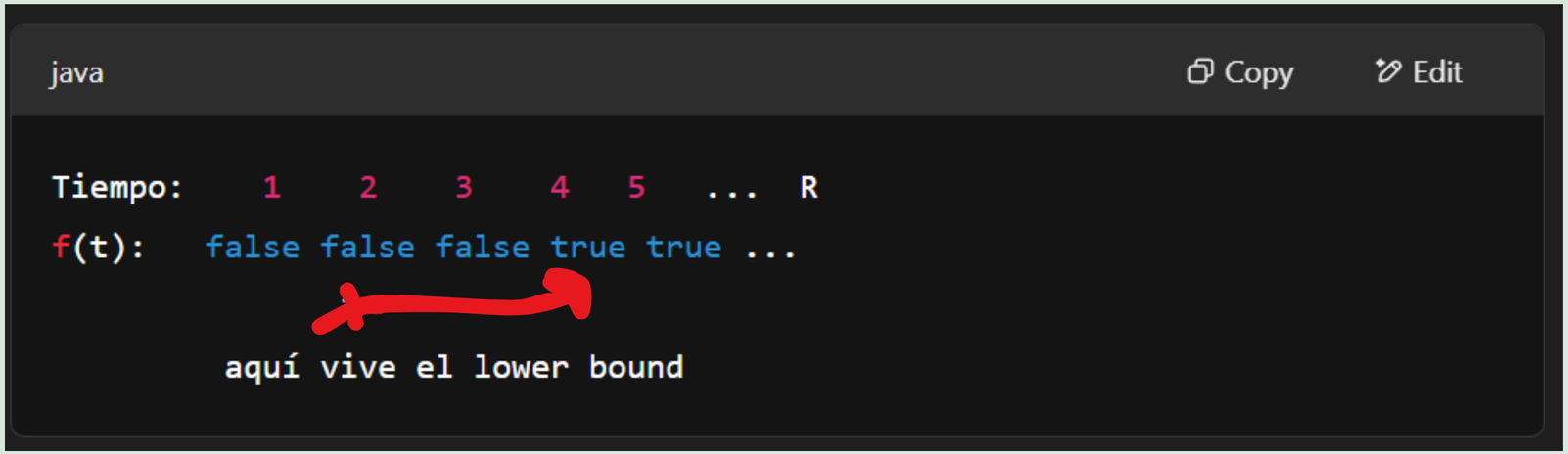
```cpp
bool f(ll t, vector<int> &maquinas, ll k) {
  ll total = 0;
  for (int i = 0; i < maquinas.size(); i++) {
    int p = maquinas[i];
    // cantidad de objetos producidos por la máquina p en t segundos,
    // podemos comprobarlo a mano
    total += t / p;
    // pequeña optimización lazy: si ya alcanzamos k, no necesitamos seguir sumando
    if (total >= k) return true;
  }
  return false;
}
// esta busqueda binaria es exactamente igual a la implementacion de busqueda binaria
// lowerbound en un arreglo, pero, fijemonos como cambia la forma en la
// que obtenemos la imagen de la funcion; en lugar de un simple a[m], es un f(m)
ll binsearch(int n, vector<int> &maquinas, ll k) {
  ll l = 1;  // mejor caso
  ll r = 1e9+1;  // peor caso
  ll res = -1;
  while (l <= r) {
    ll m = l + (r - l) / 2;
    if (f(m, maquinas, k)) {
      // encontramos un t valido (representado por m),
      // buscar uno menor aun
      res = m;
      r = m - 1;
    } else {
      l = m + 1; // buscar mas arriba
    }
  }
  return res;
}

int main() {
  int n = 3;
  vector<int> maquinas = {2, 3, 7}; // Cada máquina tarda estos segundos por objeto
  long long k = 10; // Cantidad de objetos que queremos producir
  cout << "Tiempo mínimo necesario: " << binsearch(n, maquinas, k) << " segundos" << endl;

  return 0;
}
```

¿MONOTONICIDAD?

La monotonicidad de la función se ve así:



“Hay varios puntos de tiempo donde las k piezas ya estarán producidas (o incluso más), quizá en 30 minutos (t=30) hay 4 piezas, y quizá en 20 minutos (t=20) hay 4 piezas también, entonces deducimos que existe monotonicidad (varios t pueden llevar al mismo valor en la imagen)”.

PROBLEMAS

Fácil - Codeforces 2051B

Fácil - Codeforces 1592A

Medio - Codeforces 1725B

Insano - Codeforces 2075C

Insano - Codeforces 702C

PISTAS (TEMAS)

Fácil - Codeforces 2051B

Lower bound en función, matemáticas (aritmética).

Fácil - Codeforces 1592A

Lower bound en función, greedy, sorting/implementación, matemáticas (aritmética).

Medio - Codeforces 1725B

Lower bound en función, greedy, sorting, two pointers.

Insano - Codeforces 2075C

Lower bound en arreglo, PFC, contribución, sorting.

Insano - Codeforces 702C

Lower bound en función, two pointers.

¿Y AHORA QUE?

Upsolving

Pueden seguir intentando los problemas en su casa, si consideran que ya se esforzaron un buen tiempo y no salieron, esperen otro mes para intentarlos de nuevo o vean la editorial y compréndanla.

Upper bound

Solo vimos búsqueda bínaria lowerbound (número más pequeño que cumple una condición), pero tambien existe otra versión que retorna el número más grande que cumple una condición y otra versión que retorna el número más pequeño que no cumple una condición (upper bound).

Y los decimales...

Practicar

Seguir practicando ejercicios hasta que lleguen a resolver uno de ICPC como regional LATAM 2017: Linearville

Más temas

Búsqueda binaria es base de temas como búsqueda ternaria, fenwick lowerbound o segtree + binsearch (aunque se requiere saber otros temas antes de estos 2).

Pueden estudiarlos ustedes o pedir verlos en el club.

Universidad Borcelle

MUCHAS GRACIAS

Septiembre 2030