Técnicas básicas: Parte III

CLUB DE PROGRAMACIóN COMPETITIVA UADY

Técnicas básicas

Son técnicas recurrentes que son fundamentales para la resolución de problemas básicos, y suelen ser herramientas útiles durante la implementación de problemas de mayor dificultad. En este curso se abordan 4 de estas técnicas:

Cubetas
Dos punteros
Prefix sum
Búsqueda binaria

Imaginemos que tenemos un arreglo cuyos elementos son únicamente 0 y 1, y se nos garantiza que todos los elementos son 0 hasta cierta posición, después de esa posición todos los elementos son 1.

a	0	0	0	0	0	0	1	1	1	1

Imaginemos que tenemos un arreglo cuyos elementos son únicamente 0 y 1, y se nos garantiza que todos los elementos son 0 hasta cierta posición, después de esa posición todos los elementos son 1.

¿En qué posición podemos encontrar el primer 1?

a	0	0	0	0	0	0	1	1	1	1

Imaginemos que tenemos un arreglo cuyos elementos son únicamente 0 y 1, y se nos garantiza que todos los elementos son 0 hasta cierta posición, después de esa posición todos los elementos son 1.

¿En qué posición podemos encontrar el primer 1?
Podemos contestar esta pregunta en **O(n)** recorriendo linealmente el arreglo hasta encontrar el primer elemento igual a 1.

a	0	0	0	0	0	0	1	1	1	1

Imaginemos que tenemos un arreglo cuyos elementos son únicamente 0 y 1, y se nos garantiza que todos los elementos son 0 hasta cierta posición, después de esa posición todos los elementos son 1.

¿En qué posición podemos encontrar el primer 1?
Podemos contestar esta pregunta en **O(n)** recorriendo linealmente el arreglo hasta encontrar el primer elemento igual a 1, pero veremos que esta solución es subóptima.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.

a	0	0	0	0	0	**0**	1	1	1	1

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

Busqueda binaria

La intuición principal es que si un elemento no es 1, sabemos que ningún elemento en algúna posición anterior será 1.
Similarmente podemos decir que si un elemento es 1, ningún elemento en posición posterior será el primero.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

Independientemente de en qué elemento iniciamos la busqueda, sabemos que podemos descartar todos los elementos a la derecha o a la izquierda de este simplemente conociendo su valor.

a	0	0	0	0	0	0	1	1	1	1

Busqueda binaria

Independientemente de en qué elemento iniciamos la busqueda, sabemos que podemos descartar todos los elementos a la derecha o a la izquierda de este simplemente conociendo su valor.
Dado que el elemento que buscamos puede estar en cualquier posición, resulta conveniente iniciar la busqueda exactamente a la mitad del arreglo, dado que independientemente del valor del elemento sabemos que descartaremos la mitad del espacio de busqueda.

a	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

Busqueda binaria

Independientemente de en qué elemento iniciamos la busqueda, sabemos que podemos descartar todos los elementos a la derecha o a la izquierda de este simplemente conociendo su valor.
Dado que el elemento que buscamos puede estar en cualquier posición, resulta conveniente iniciar la busqueda exactamente a la mitad del arreglo, dado que independientemente del valor del elemento sabemos que descartaremos la mitad del espacio de busqueda.

a	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

Busqueda binaria

Independientemente de en qué elemento iniciamos la busqueda, sabemos que podemos descartar todos los elementos a la derecha o a la izquierda de este simplemente conociendo su valor.
Dado que el elemento que buscamos puede estar en cualquier posición, resulta conveniente iniciar la busqueda exactamente a la mitad del arreglo, dado que independientemente del valor del elemento sabemos que descartaremos la mitad del espacio de busqueda.

a

0	0	1	1	1	1	1	1

Busqueda binaria

Independientemente de en qué elemento iniciamos la busqueda, sabemos que podemos descartar todos los elementos a la derecha o a la izquierda de este simplemente conociendo su valor.
Dado que el elemento que buscamos puede estar en cualquier posición, resulta conveniente iniciar la busqueda exactamente a la mitad del arreglo, dado que independientemente del valor del elemento sabemos que descartaremos la mitad del espacio de busqueda.

a	0	0	1	1	1	1	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	0	0	1	1	1	1	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	0	0	1	1	1	1	1	1

CLUB DE PROGRAMACIóN COMPETITIVA UADY

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	0	0	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a				
	0	0	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	0	0	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	0	0	1	1

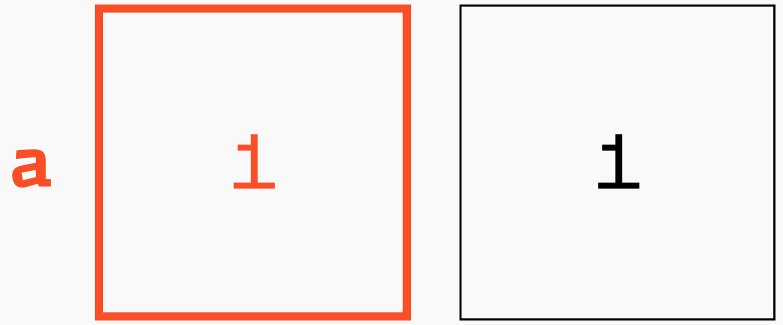
Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1



Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	1	1

Busqueda binaria

Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	1

Busqueda binaria

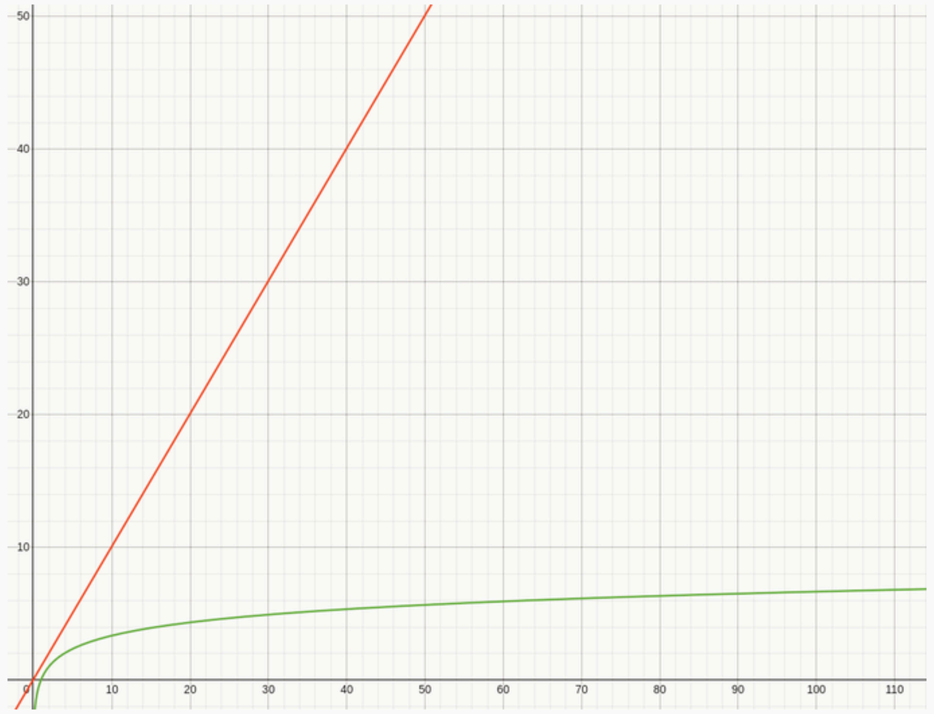
Ahora es facil ver que el problema es exáctamente el mismo, pero con la mitad de elementos con los que iniciamos.
Es facil ver que si repetimos este proceso de eliminar la mitad de los elementos no tardaremos en quedarnos con solo 1

a	1

Busqueda binaria

Es un algoritmo de búsqueda que encuentra la posición de un valor en un arreglo ordenado. Dado que en cada iteración el espacio de búsqueda se reduce a la mitad, podemos ver que para una entrada de tamaño **n**, realizar una búsqueda binaria tiene una complejidad **O(log n)**.

Esta complejidad nos permite valores de **n** prácticamente ilimitados, con la condición de que nuestro arreglo inicial ya esté ordenado antes de iniciar la búsqueda.



Podemos notar que el crecimiento de **O(log n)** (en verde) es extremadamente menos acelerado que **O(n)** (en rojo).

Esto es particularmente notorio para valores de **n** muy grandes. Mientras que **log(8) = 3**, **log(1E18) ≈ 60**

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

Si imaginamos que todos los elementos menores a **x** son 0 y todos los elementos mayores o iguales son 1 nuestro problema es exáctamente el mismo que el anterior

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

Si imaginamos que todos los elementos menores a **x** son 0 y todos los elementos mayores o iguales son 1 nuestro problema es exáctamente el mismo que el anterior

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66
	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1

$x = 12$

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

Si imaginamos que todos los elementos menores a **x** son 0 y todos los elementos mayores o iguales son 1 nuestro problema es exáctamente el mismo que el anterior

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66
	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1

x = 25

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

Si imaginamos que todos los elementos menores a **x** son 0 y todos los elementos mayores o iguales son 1 nuestro problema es exáctamente el mismo que el anterior

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

$x = 7$

Supongamos que contamos con un arreglo **a** ordenado de tamaño **n** y queremos contestar **q** preguntas del estilo: ¿Cual es la posición del primer elemento mayor o igual a **x**?

Si imaginamos que todos los elementos menores a **x** son 0 y todos los elementos mayores o iguales son 1 nuestro problema es exáctamente el mismo que el anterior

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

x = 50

CLUB DE PROGRAMACIÓN COMPETITIVA UADY

	l																r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	-	0	15	-

Implementación

	l																r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	-	0	15	-

Implementación

CLUB DE
PROGRAMACIÓN
COMPETITIVA
UADY

	l							mid									r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	19	0	15	7

Implementación

CLUB DE PROGRAMACIóN COMPETITIVA UADY

	l							mid									r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	19	0	15	7

Implementación

								mid	l								r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	19	8	15	7

Implementación

CLUB DE PROGRAMACIÓN COMPETITIVA UADY

								mid	l								r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	19	8	15	7

Implementación

CLUB DE PROGRAMACIÓN COMPETITIVA
UADY

									l			mid				r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	37	8	15	11

Implementación

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

									l			mid					r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	37	8	15	11

Implementación

CLUB DE PROGRAMACIóN COMPETITIVA UADY

	mid	l			r

a: 1 3 6 9 11 13 18 19 21 27 34 37 49 52 57 66

(mid = 37, l = 49, r after 66)

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	37	12	15	11

Implementación

											mid	l				r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```cpp
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	37	12	15	11

Implementación

CLUB DE PROGRAMACIóN COMPETITIVA
UADY

													l	mid			r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66	

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	52	12	15	13

Implementación

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

												l	mid			r
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	52	12	15	13

Implementación

												l	r mid		
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	52	12	13	13

Implementación

CLUB DE PROGRAMACIÓN COMPETITIVA UADY

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66
													l	r, mid		

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	52	12	13	13

Implementación

CLUB DE PROGRAMACIóN COMPETITIVA **UADY**

												mid l	r		
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	49	12	13	12

Implementación

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

													mid l	r		
a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	49	12	13	12

Implementación

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY

r
mid
l

a	1	3	6	9	11	13	18	19	21	27	34	37	49	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	49	12	12	12

Implementación

													r mid l			
a	1	3	6	9	11	13	18	19	21	27	34	37	**49**	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

x	a[mid]	l	r	mid
41	49	12	12	12

Implementación

r
mid
l

a	1	3	6	9	11	13	18	19	21	27	34	37	$\boxed{49}$	52	57	66

```
int l = 0, r = n, mid;

while(l < r){
    mid = (l+r)/2;
    if(a[mid] >= x){
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

l

12

Implementación

En general todos los problemas de búsqueda binaria se reducen al primer problema:

Existe un valor para el que todos los elementos anteriores no se cumple cierta condición (evalúan a 0), y todos los elementos posteriores cumplen con esta (evalúan a 1), por lo que simplemente tenemos que cambiar el criterio con el cual separamos el el espacio de búsqueda (no necesariamente un arreglo!) y podremos realizarla siempre y cuando el espacio esté ordenado según esta condicional.

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Ejemplo del espácio de búsqueda no siendo un arreglo:

Tenemos **n** preguntas del estilo:

Dado un número **x**, encuentra el primer entero **y** tal que tal que la suma de los primeros **y** números naturales es mayor o igual a **x**

Entrada:

En la primera línea un número **n**, el número de preguntas. **1 <= n <= 2e5.**
Las siguientes **n** lineas, enteros **x.** **1 <= x <= 1e9.**

Entrada de ejemplo	**Salida esperada**
5	4
10	7
28	8
36	14
100	1640
1345312	

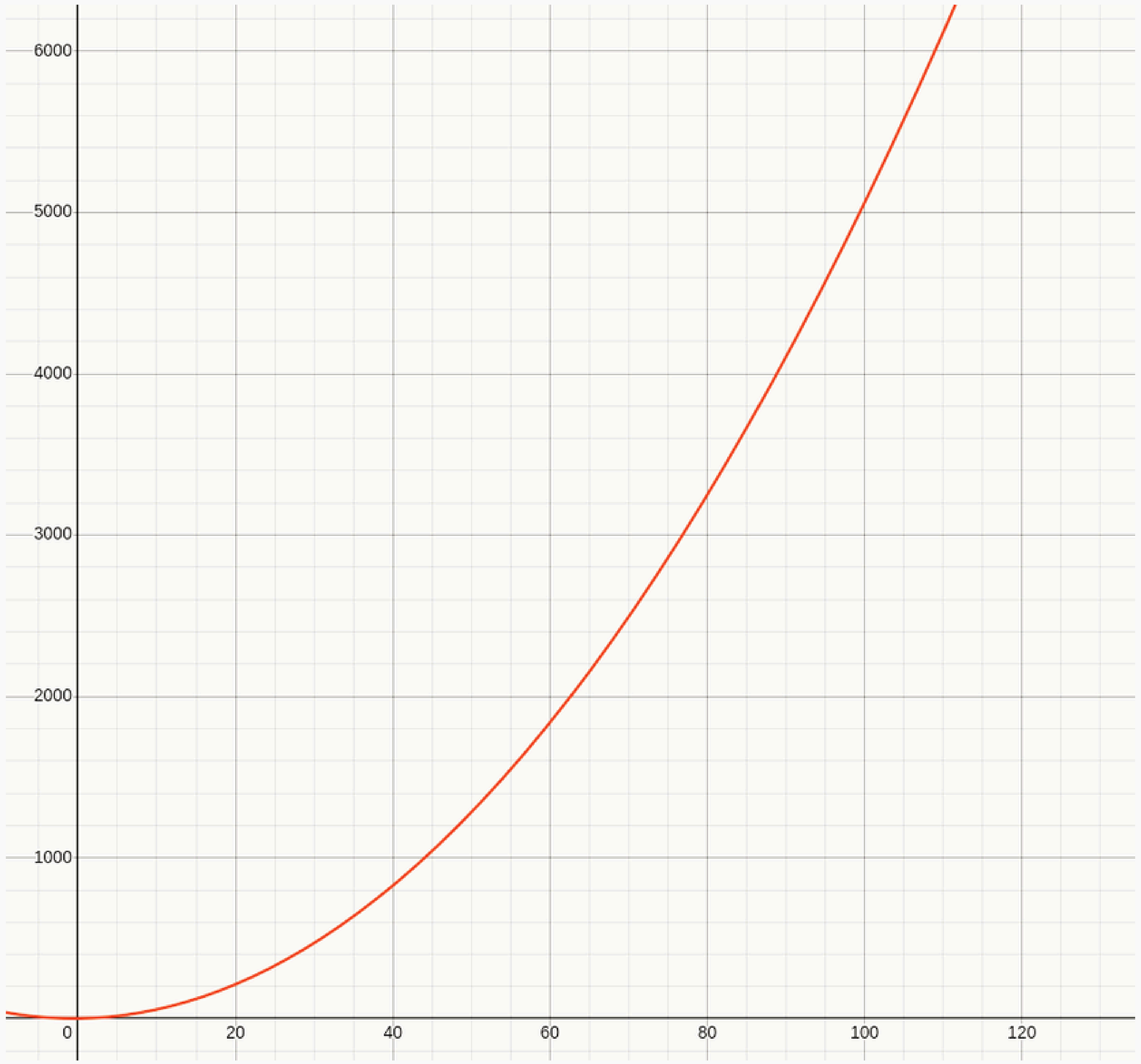
Observaciones

1. La suma de los primeros **x** números naturales es igual a **x(x+1)/2,** conocido como sumatoria de gauss, que llamaremos **g(x)**.

CLUB DE PROGRAMACIÓN COMPETITIVA UADY

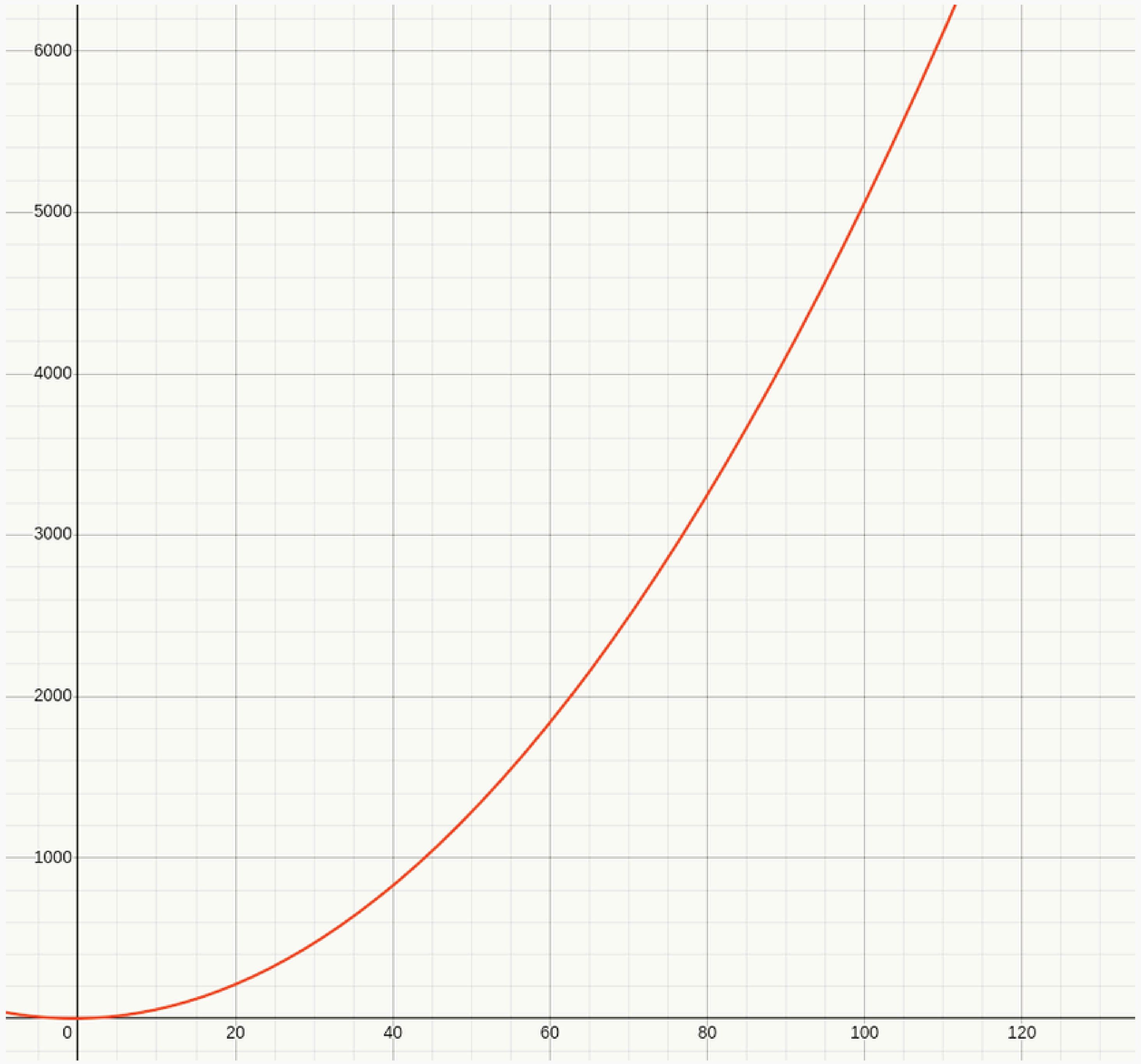
Observaciones

1. La suma de los primeros **x** números naturales es igual a **x(x+1)/2,** conocido como sumatoria de gauss, que llamaremos **g(x)**.
2. **g(x)** es una función no decreciente para todo **x >= 0**, por lo que podemos decir que el contradominio está ordenado para todos los naturales.



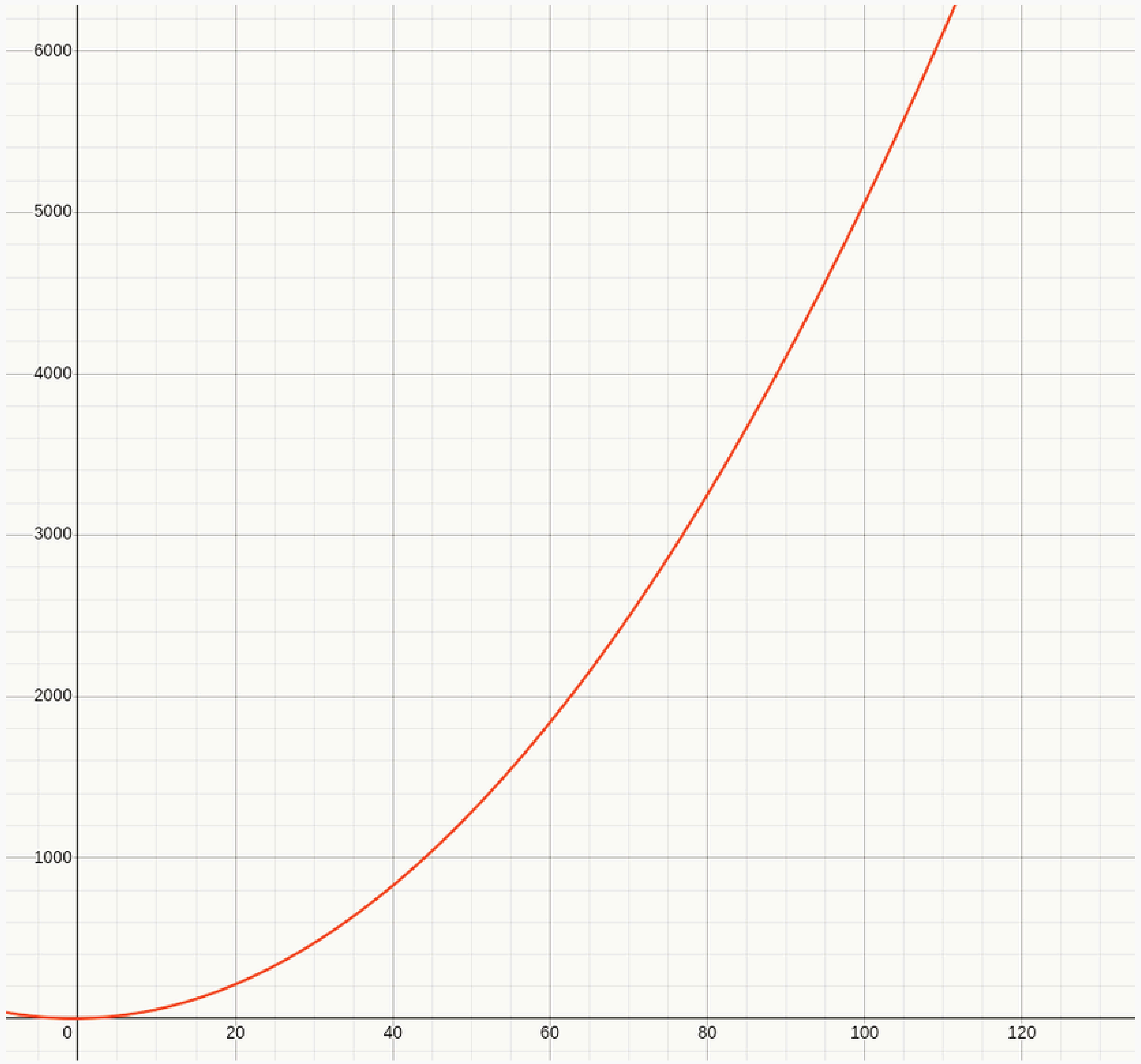
Observaciones

1. La suma de los primeros **x** números naturales es igual a **x(x+1)/2,** conocido como sumatoria de gauss, que llamaremos **g(x)**.
2. **g(x)** es una función no decreciente para todo **x >= 0**, por lo que podemos decir que el contradominio está ordenado para todos los naturales.
3. Dado que el espácio de búsqueda está ordenado podemos aplicar una búsqueda binaria sobre todos los naturales.



Observaciones

1. La suma de los primeros **x** números naturales es igual a **x(x+1)/2,** conocido como sumatoria de gauss, que llamaremos **g(x)**.
2. **g(x)** es una función no decreciente para todo **x >= 0**, por lo que podemos decir que el contradominio está ordenado para todos los naturales.
3. Dado que el espácio de búsqueda está ordenado podemos aplicar una búsqueda binaria sobre todos los naturales.
4. Podemos resolver en **O(n log x)** el problema



```cpp
ll gauss(ll x){
    return (x*(x+1))/2;
}

ll l = 0, r = 1E9, mid;

while(l < r){
     mid = (l+r)/2;
     if(gauss(mid) >= x){
          r = mid;
     } else {
          l = mid + 1;
     }
}

cout << l << "\n";
```

En esta implementación, el "arreglo" sobre el que hacemos la búsqueda binaria, son los valores de **g(x)** para todo **0 <= x <= 1E9,** que no necesitamos generar explícitamente dado que conocemos la función.

En general podemos emplear esta técnica para encontrar **x** solo conociendo **f(x),** o en otras palabras encontrar la inversa de toda función, siempre que **f(x)** sea **monótona** en el intervalo en el que deseamos hacer la búsqueda, con complejidad **O(log x)**

upper_bound, lower_bound

En la standard template library tenemos contenedores ordenados, por lo que da pie a que podamos realizar este tipo de búsqueda con la misma complejidad **O(log n)**. Por suerte, estas funciones ya están implementadas como métodos de estos contenedores:

```
std::set::lower_bound(5)            // Primer elemento mayor o igual a x     {1, 3, 5, 7, 8}
std::set::upper_bound(5)            // Primer elemento mayor a x             {1, 3, 5, 7, 8}
std::multiset::lower_bound(5)       // Primer elemento mayor o igual a x     {1, 5, 5, 5, 8}
std::multiset::upper_bound(5)       // Primer elemento mayor a x             {1, 5, 5, 5, 8}
```

Para los contenedores de tipo map el funcionamiento es el mismo.
En caso de no encontrarse el elemento se retorna el `end()`

CLUB DE
PROGRAMACIóN
COMPETITIVA
UADY