Complejidad de Algoritmos y Jueces Online

Ariel Leonardo Fideleff

Entrenamiento en Programación Competitiva
Facultad de Ingeniería - Universidad Nacional de Jujuy (UNJu)

30 de mayo del 2025

Hola

1. Introducción

2. Complejidad computacional
   - Un ejemplo
   - Definición
   - Chau constantes (o casi)
   - Algunos órdenes de complejidad comunes
   - Estimando el tiempo de ejecución
   - Análisis amortizado

3. Jueces Online
   - Introducción
   - omegaUp: lo básico
   - Veredictos y otros jueces

Empecemos por un ejemplo

Empecemos por un ejemplo

Veamos el siguiente código en C++:

Empecemos por un ejemplo

Veamos el siguiente código en C++:

```cpp
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]); // Intercambia los números
}
```

Empecemos por un ejemplo

Veamos el siguiente código en C++:

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]); // Intercambia los números
}
```

¿Qué hace?

Empecemos por un ejemplo

Veamos el siguiente código en C++:

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]); // Intercambia los números
}
```

¿Qué hace? Ordena una lista de enteros de menor a mayor!

Empecemos por un ejemplo

Veamos el siguiente código en C++:

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]); // Intercambia los números
}
```

¿Qué hace? Ordena una lista de enteros de menor a mayor!

En efecto, en cada vuelta del primer `for`, se elige el elemento más chico entre ese y los que le siguen.

Empecemos por un ejemplo

Veamos el siguiente código en C++:

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]); // Intercambia los números
}
```

¿Qué hace? Ordena una lista de enteros de menor a mayor!

En efecto, en cada vuelta del primer `for`, se elige el elemento más chico entre ese y los que le siguen.
Este algoritmo es conocido, se llama **selection sort**.

Otro algoritmo para la misma cosa

Ordenar números es fácil, no?

Otro algoritmo para la misma cosa

Ordenar números es fácil, no? Y tampoco es la única forma...

Otro algoritmo para la misma cosa

Ordenar números es fácil, no? Y tampoco es la única forma...

```
void merge(vector<int> &arr, int l, int mid, int r) {
    int asz = mid-l, bsz = r-mid;
    vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
        a[i] = arr[l+i];
    for (int i = 0; i < bsz; i++)
        b[i] = arr[mid+i];

    int aidx = 0, bidx = 0, idx = l;
    while (aidx < asz && bidx < bsz)
        if (a[aidx] <= b[bidx]) {
            arr[idx] = a[aidx];
            idx++, aidx++;
        }
        else {
            arr[idx] = b[bidx];
            idx++, bidx++;
        }

    while (aidx < asz) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    while (bidx < bsz) {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
}
```

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}

void sort(vector<int> &arr, int n) {
    split(arr,0,n);
}
```

Otro algoritmo para la misma cosa

Ordenar números es fácil, no? Y tampoco es la única forma...

```cpp
void merge(vector<int> &arr, int l, int mid, int r) {
    int asz = mid-l, bsz = r-mid;
    vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
        a[i] = arr[l+i];
    for (int i = 0; i < bsz; i++)
        b[i] = arr[mid+i];

    int aidx = 0, bidx = 0, idx = l;
    while (aidx < asz && bidx < bsz)
        if (a[aidx] <= b[bidx]) {
            arr[idx] = a[aidx];
            idx++, aidx++;
        }
        else {
            arr[idx] = b[bidx];
            idx++, bidx++;
        }

    while (aidx < asz) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    while (bidx < bsz) {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
}
```

```cpp
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}

void sort(vector<int> &arr, int n) {
    split(arr,0,n);
}
```

Mucho código...

Otro algoritmo para la misma cosa

Ordenar números es fácil, no? Y tampoco es la única forma...

```
void merge(vector<int> &arr, int l, int mid, int r) {
    int asz = mid-l, bsz = r-mid;
    vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
        a[i] = arr[l+i];
    for (int i = 0; i < bsz; i++)
        b[i] = arr[mid+i];

    int aidx = 0, bidx = 0, idx = l;
    while (aidx < asz && bidx < bsz)
        if (a[aidx] <= b[bidx]) {
            arr[idx] = a[aidx];
            idx++, aidx++;
        }
        else {
            arr[idx] = b[bidx];
            idx++, bidx++;
        }

    while (aidx < asz) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    while (bidx < bsz) {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
}
```

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}

void sort(vector<int> &arr, int n) {
    split(arr,0,n);
}
```

Mucho código... pero también ordena una lista de enteros de menor a mayor!

Otro algoritmo para la misma cosa

Ordenar números es fácil, no? Y tampoco es la única forma...

```
void merge(vector<int> &arr, int l, int mid, int r) {
    int asz = mid-l, bsz = r-mid;
    vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
        a[i] = arr[l+i];
    for (int i = 0; i < bsz; i++)
        b[i] = arr[mid+i];

    int aidx = 0, bidx = 0, idx = l;
    while (aidx < asz && bidx < bsz)
        if (a[aidx] <= b[bidx]) {
            arr[idx] = a[aidx];
            idx++, aidx++;
        }
        else {
            arr[idx] = b[bidx];
            idx++, bidx++;
        }

    while (aidx < asz) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    while (bidx < bsz) {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
}
```

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}

void sort(vector<int> &arr, int n) {
    split(arr,0,n);
}
```

Mucho código... pero también ordena una lista de enteros de menor a mayor!
Este algoritmo también es conocido, se llama **merge sort**.

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?

Para entenderlo, usaremos el comando `time` de Linux.

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?

Para entenderlo, usaremos el comando `time` de Linux.

Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?

Para entenderlo, usaremos el comando `time` de Linux.

Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

```
[ariel@arch-pve code]$ time ./msort

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

(a) Merge Sort

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?
Para entenderlo, usaremos el comando `time` de Linux.
Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

```
[ariel@arch-pve code]$ time ./msort

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

(a) Merge Sort

```
[ariel@arch-pve code]$ time ./ssort

real    0m45.235s
user    0m45.133s
sys     0m0.000s
```

(b) Selection Sort

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?

Para entenderlo, usaremos el comando `time` de Linux.

Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

```
[ariel@arch-pve code]$ time ./msort

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

(a) Merge Sort

```
[ariel@arch-pve code]$ time ./ssort

real    0m45.235s
user    0m45.133s
sys     0m0.000s
```

(b) Selection Sort

Parece que una de las dos se toma su tiempo.

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?
Para entenderlo, usaremos el comando `time` de Linux.
Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

```
[ariel@arch-pve code]$ time ./msort

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

(a) Merge Sort

```
[ariel@arch-pve code]$ time ./ssort

real    0m45.235s
user    0m45.133s
sys     0m0.000s
```

(b) Selection Sort

Parece que una de las dos se toma su tiempo.
Entonces incluso teniendo dos algoritmos que logran exactamente lo mismo, hay diferencias que van más allá de sus resultados.

La liebre versus la tortuga

Pero pará, si *selection sort* ordena una lista, es corto y fácil de entender, ¿para qué usaría *merge sort*?
Para entenderlo, usaremos el comando `time` de Linux.
Supongamos que tenemos una lista aleatoria de 10^5 enteros... (sí, son muchos enteros)

```
[ariel@arch-pve code]$ time ./msort

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

(a) Merge Sort

```
[ariel@arch-pve code]$ time ./ssort

real    0m45.235s
user    0m45.133s
sys     0m0.000s
```

(b) Selection Sort

Parece que una de las dos se toma su tiempo.
Entonces incluso teniendo dos algoritmos que logran exactamente lo mismo, hay diferencias que van más allá de sus resultados.
Al final tanto código sí servía para algo, no?

$\mathcal{O}$ qué!?

En Programación Competitiva nos gusta que las cosas sean rápidas.

$\mathcal{O}$ qué!?

En Programación Competitiva nos gusta que las cosas sean rápidas.
Uno de los desafíos del deporte no es sólo lograr resolver lo que nos piden, sino también rápido (nosotros en pensar, y el programa en computar).

$\mathcal{O}$ qué!?

En Programación Competitiva nos gusta que las cosas sean rápidas.
Uno de los desafíos del deporte no es sólo lograr resolver lo que nos piden, sino también rápido (nosotros en pensar, y el programa en computar).
Así, viene útil poder identificar "qué tan rápido" es un programa con mirarlo un poco.

$\mathcal{O}$ qué!?

En Programación Competitiva nos gusta que las cosas sean rápidas.
Uno de los desafíos del deporte no es sólo lograr resolver lo que nos piden, sino también rápido (nosotros en pensar, y el programa en computar).
Así, viene útil poder identificar "qué tan rápido" es un programa con mirarlo un poco.
Para ello se usa frecuentemente la **notación $\mathcal{O}$**, también llamada **notación O grande**, **cota superior asintótica**, o bien como le decimos nosotros "el programa corre en O en [lo que haya dentro de los paréntesis]".

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Consideremos un programa que recibe en su entrada una cantidad de datos n.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Consideremos un programa que recibe en su entrada una cantidad de datos n. Si a partir de un "n suficientemente grande" la cantidad de operaciones que ejecuta el programa es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que corre en $\mathcal{O}(n)$.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Consideremos un programa que recibe en su entrada una cantidad de datos n. Si a partir de un "n suficientemente grande" la cantidad de operaciones que ejecuta el programa es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que corre en $\mathcal{O}(n)$.

Veamos un ejemplo.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Consideremos un programa que recibe en su entrada una cantidad de datos n. Si a partir de un "n suficientemente grande" la cantidad de operaciones que ejecuta el programa es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que corre en $\mathcal{O}(n)$.

Veamos un ejemplo... pero primero reescribamos un poco y enmarquemos la definición de antes para tenerla como referencia.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Veamos un ejemplo... pero primero reescribamos un poco y enmarquemos la definición de antes para tenerla como referencia.

¿Cómo L$\mathcal{O}$ usamos?

Buenísimo entonces, tenemos una herramienta para saber qué tan rápido es un programa. Igual... cómo la usamos?

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Veamos un ejemplo... pero primero reescribamos un poco y enmarquemos la definición de antes para tenerla como referencia. Ahí va.

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];

cout << sum;
```

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];

cout << sum;
```

$\rightarrow$ n operaciones (leemos n números)

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)      → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)      → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;
```

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)      → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)      → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;
```

Hicimos entonces $2n$ operaciones. O sea, $c * n$ con $c = 2$

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)      → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)      → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;
```

Hicimos entonces $2n$ operaciones. O sea, $c * n$ con $c = 2$ ... no?

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;                  → 1 operación (leemos 1 número)
for (int i = 0; i < n; i++)       → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)       → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;                      → 1 operación (imprimimos 1 número)
```

Hicimos entonces $2n$ operaciones. O sea, $c * n$ con $c = 2$ ... no?

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

Consideremos un programa que lee los números de una lista y los suma:

```
int arr[1000];
int n; cin >> n;                  → 1 operación (leemos 1 número)
for (int i = 0; i < n; i++)       → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)       → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;                      → 1 operación (imprimimos 1 número)
```

Hicimos entonces $2n + 2$ operaciones. $2n + 2 \leq 2n + n = 3n$ para $n \geq 2$.

Un programa que corre en $\mathcal{O}(n)$

Definición (notación $\mathcal{O}$, informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un n suficientemente grande la cantidad de operaciones que ejecuta es **a lo sumo** $c * n$, donde c es una constante *positiva*, entonces decimos que el programa corre en $\mathcal{O}(n)$.

```
int arr[1000];
int n; cin >> n;                  → 1 operación (leemos 1 número)
for (int i = 0; i < n; i++)       → n operaciones (leemos n números)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)       → n operaciones (sumamos n números)
    sum += arr[i];

cout << sum;                      → 1 operación (imprimimos 1 número)
```

Hicimos entonces $2n + 2$ operaciones. $2n + 2 \leq 2n + n = 3n$ para $n \geq 2$. Por lo tanto, este programa corre en $\mathcal{O}(n)$, pues ejecuta a lo sumo $3n$ operaciones a partir de un n "suficientemente grande".

Definición formal (para los más curiosos)

En realidad la notación $\mathcal{O}$ nos permite acotar una función cuando su argumento tiende a infinito.

Definición formal (para los más curiosos)

En realidad la notación $\mathcal{O}$ nos permite acotar una función cuando su argumento tiende a infinito.
Cuando vimos que el programa "corría en $\mathcal{O}(n)$" definimos implícitamente una función $f(n) = 2n + 2$ a la cual le atribuímos describir la cantidad de operaciones que ejecuta en función de la entrada.

Definición formal (para los más curiosos)

En realidad la notación $\mathcal{O}$ nos permite acotar una función cuando su argumento tiende a infinito.
Cuando vimos que el programa "corría en $\mathcal{O}(n)$" definimos implícitamente una función $f(n) = 2n + 2$ a la cual le atribuímos describir la cantidad de operaciones que ejecuta en función de la entrada.

Definición (notación $\mathcal{O}$)

Sean funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que $f \in \mathcal{O}(g)$ si $\exists\, c \in \mathbb{R}_0^+, n_0 \in \mathbb{N}$ tal que $0 \leq f(n) \leq c.g(n)$, $\forall n \geq n_0$.

Definición formal (para los más curiosos)

En realidad la notación $\mathcal{O}$ nos permite acotar una función cuando su argumento tiende a infinito.
Cuando vimos que el programa "corría en $\mathcal{O}(n)$" definimos implícitamente una función $f(n) = 2n + 2$ a la cual le atribuímos describir la cantidad de operaciones que ejecuta en función de la entrada.

Definición (notación $\mathcal{O}$)

Sean funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que $f \in \mathcal{O}(g)$ si $\exists\, c \in \mathbb{R}_0^+, n_0 \in \mathbb{N}$ tal que $0 \leq f(n) \leq c.g(n),\ \forall n \geq n_0$.

Es decir, si graficáramos ambas funciones $f(n) = 2n + 2$ y $g(n) = n$, existe $n_0 = 2$ natural a partir del cual la función f siempre tiene valores menores o iguales que la función $c.g = 3g = 3n$.

Definición formal (para los más curiosos)

En realidad la notación $\mathcal{O}$ nos permite acotar una función cuando su argumento tiende a infinito.
Cuando vimos que el programa "corría en $\mathcal{O}(n)$" definimos implícitamente una función $f(n) = 2n + 2$ a la cual le atribuímos describir la cantidad de operaciones que ejecuta en función de la entrada.

Definición (notación $\mathcal{O}$)

Sean funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que $f \in \mathcal{O}(g)$ si $\exists\, c \in \mathbb{R}_0^+, n_0 \in \mathbb{N}$ tal que $0 \leq f(n) \leq c.g(n),\ \forall n \geq n_0$.

Es decir, si graficáramos ambas funciones $f(n) = 2n + 2$ y $g(n) = n$, existe $n_0 = 2$ natural a partir del cual la función f siempre tiene valores menores o iguales que la función $c.g = 3g = 3n$. La función f está **acotada** por $c.g$.

Una constante no deja de ser constante

Como vimos en el ejemplo de la sección anterior, las constantes no afectan a la cota $\mathcal{O}$ de un programa.

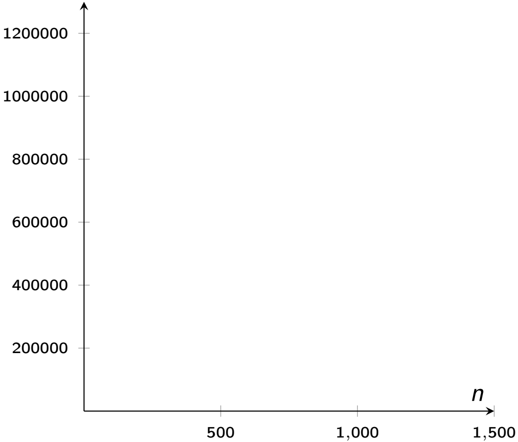
Una constante no deja de ser constante

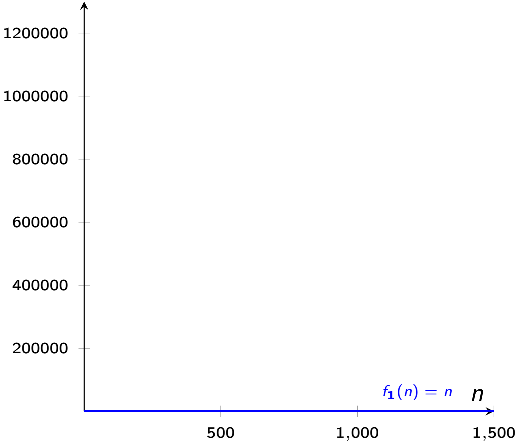
Como vimos en el ejemplo de la sección anterior, las constantes no afectan a la cota $\mathcal{O}$ de un programa.
Si bien técnicamente por definición podríamos decir que un programa "corre en $\mathcal{O}(2n)$" en vez de decir que corre en $\mathcal{O}(n)$, no tiene mucho sentido hacerlo porque lo que más afecta a la rapidez de nuestro programa para una entrada suficientemente grande es la cantidad de operaciones que realiza respecto a su tamaño.

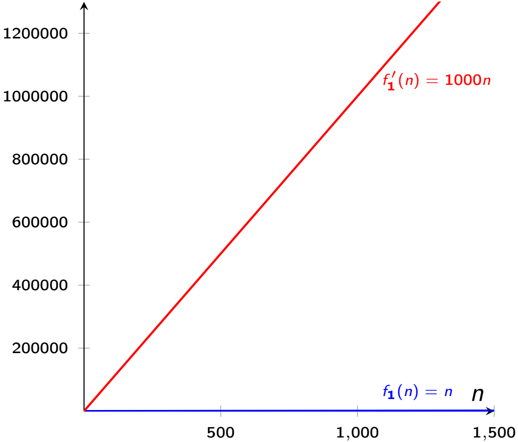
Una constante no deja de ser constante

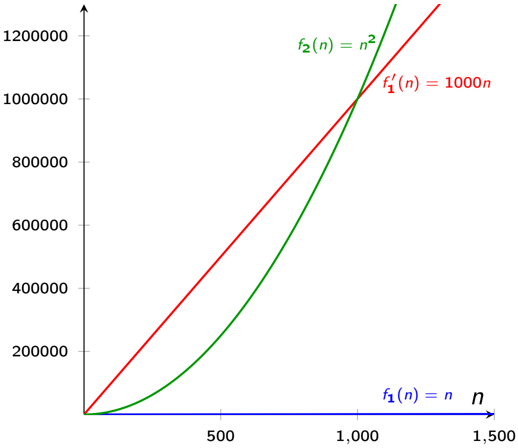
Como vimos en el ejemplo de la sección anterior, las constantes no afectan a la cota $\mathcal{O}$ de un programa.
Si bien técnicamente por definición podríamos decir que un programa "corre en $\mathcal{O}(2n)$" en vez de decir que corre en $\mathcal{O}(n)$, no tiene mucho sentido hacerlo porque lo que más afecta a la rapidez de nuestro programa para una entrada suficientemente grande es la cantidad de operaciones que realiza respecto a su tamaño.

Podemos pensarlo de la siguiente forma: si el factor constante de nuestro programa que corre en $\mathcal{O}(n)$ es muy alta, por ejemplo $c = 1000$, entonces a partir de $n = 1000$, un programa que corre en $\mathcal{O}(n^2)$ realiza más operaciones que el primero:


1200000
1000000
800000
600000
400000
200000
500
1,000
1,500
n


1200000
1000000
800000
600000
400000
200000
$f_1(n) = n$
n
500
1,000
1,500


1200000
1000000
800000
600000
400000
200000
$f_1'(n) = 1000n$
$f_1(n) = n$
n
500
1,000
1,500


1200000
1000000
800000
600000
400000
200000
500
1,000
1,500
n
$f_2(n) = n^2$
$f'_1(n) = 1000n$
$f_1(n) = n$

Resolver problemas va más allá de las constantes

Así, en la práctica observamos que:

Resolver problemas va más allá de las constantes

Así, en la práctica observamos que:

- Normalmente el factor constante de una solución no afecta su correctitud a la hora de ser evaluada.

Resolver problemas va más allá de las constantes

Así, en la práctica observamos que:

- Normalmente el factor constante de una solución no afecta su correctitud a la hora de ser evaluada.
- A la hora de comparar potenciales soluciones a un problema, se lo suele hacer sin tener en cuenta las constantes, pues éstas están más sujetas a detalles de la implementación de una solución, que a diferentes ideas para resolver un problema.

Resolver problemas va más allá de las constantes

Así, en la práctica observamos que:

- Normalmente el factor constante de una solución no afecta su correctitud a la hora de ser evaluada.
- A la hora de comparar potenciales soluciones a un problema, se lo suele hacer sin tener en cuenta las constantes, pues éstas están más sujetas a detalles de la implementación de una solución, que a diferentes ideas para resolver un problema.
- Reducir el factor constante utilizado por un programa suele tratarse de hacer pequeñas modificaciones al código sin cambiar la idea fundamental de la solución que implementa.

Resolver problemas va más allá de las constantes

Así, en la práctica observamos que:

- Normalmente el factor constante de una solución no afecta su correctitud a la hora de ser evaluada.
- A la hora de comparar potenciales soluciones a un problema, se lo suele hacer sin tener en cuenta las constantes, pues éstas están más sujetas a detalles de la implementación de una solución, que a diferentes ideas para resolver un problema.
- Reducir el factor constante utilizado por un programa suele tratarse de hacer pequeñas modificaciones al código sin cambiar la idea fundamental de la solución que implementa.
- Cambios en la complejidad (la cantidad de recursos utilizados por el programa) más significativos que los dados por una constante, suelen estar dados por observaciones más importantes y cambios más profundos a la idea para resolver el problema.

Siempre hay un pero

Por supuesto, hay excepciones, y puede pasar que en algunos casos el límite de tiempo sea algo ajustado, o cuando una solución tenga un factor de constante muy grande, éste sea factor que pueda decidir entre si una solución es considerada correcta o incorrecta por el sistema automático de evaluación.

Siempre hay un pero

Por supuesto, hay excepciones, y puede pasar que en algunos casos el límite de tiempo sea algo ajustado, o cuando una solución tenga un factor de constante muy grande, éste sea factor que pueda decidir entre si una solución es considerada correcta o incorrecta por el sistema automático de evaluación.

Más aún, en algunos casos muy raros puede ser la intención del/los autor/es de un problema en que reducir la constante sea necesario para resolverlo (investigar `std::bitset`).

Siempre hay un pero

Por supuesto, hay excepciones, y puede pasar que en algunos casos el límite de tiempo sea algo ajustado, o cuando una solución tenga un factor de constante muy grande, éste sea factor que pueda decidir entre si una solución es considerada correcta o incorrecta por el sistema automático de evaluación.

Más aún, en algunos casos muy raros puede ser la intención del/los autor/es de un problema en que reducir la constante sea necesario para resolverlo (investigar `std::bitset`).

A pesar de todo esto, la conclusión que se deben llevar es que es bueno saber qué es el factor constante de un programa, pero no es esencial reducirlo para resolver problemas, especialmente cuando uno empieza (en los casos más sencillos hasta el compilador los optimiza auutomáticamente!).

Los beneficios de ignorar constantes

Volviendo al ejemplo anterior de la suma de números, se vuelve más sencillo analizar la cota superior asintótica del programa.

Los beneficios de ignorar constantes

Volviendo al ejemplo anterior de la suma de números, se vuelve más sencillo analizar la cota superior asintótica del programa.

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];

cout << sum;
```

Los beneficios de ignorar constantes

Volviendo al ejemplo anterior de la suma de números, se vuelve más sencillo analizar la cota superior asintótica del programa.

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];

cout << sum;
```

$\longrightarrow$ n operaciones

Los beneficios de ignorar constantes

Volviendo al ejemplo anterior de la suma de números, se vuelve más sencillo analizar la cota superior asintótica del programa.

```
int arr[1000];
int n; cin >> n;
for (int i = 0; i < n; i++)   → n operaciones
    cin >> arr[i];

int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];

cout << sum;
```

Listo! Como vemos que en ningún otro lugar hacemos más de n operaciones, cualquier otra cantidad de operaciones que contemos será múltiplo de n, o bien una constante, los cuales no modificarán la cota $\mathcal{O}$ del programa.

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.
- Llamamos **complejidad asintótica computacional** al uso de *análisis asintótico* (como determinar la cota asintótica $\mathcal{O}$ con la que corre un programa) para estimar la complejidad computacional de un programa.

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.
- Llamamos **complejidad asintótica computacional** al uso de *análisis asintótico* (como determinar la cota asintótica $\mathcal{O}$ con la que corre un programa) para estimar la complejidad computacional de un programa.
- Otra forma de decir que un programa "corre en $\mathcal{O}(n)$" es decir que es del **orden** $\mathcal{O}(n)$ (formalmente, si $f \in \mathcal{O}(g(x))$, entonces $f(x)$ es del orden de $g(x)$).

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.
- Llamamos **complejidad asintótica computacional** al uso de *análisis asintótico* (como determinar la cota asintótica $\mathcal{O}$ con la que corre un programa) para estimar la complejidad computacional de un programa.
- Otra forma de decir que un programa "corre en $\mathcal{O}(n)$" es decir que es del **orden** $\mathcal{O}(n)$ (formalmente, si $f \in \mathcal{O}(g(x))$, entonces $f(x)$ es del orden de $g(x)$).

Así, como vimos que las constantes no son relevantes para el análisis de la cota $\mathcal{O}$, existen ciertos *órdenes de complejidad* comunes para los programas que **no** varían por constantes.

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.
- Llamamos **complejidad asintótica computacional** al uso de *análisis asintótico* (como determinar la cota asintótica $\mathcal{O}$ con la que corre un programa) para estimar la complejidad computacional de un programa.
- Otra forma de decir que un programa "corre en $\mathcal{O}(n)$" es decir que es del **orden** $\mathcal{O}(n)$ (formalmente, si $f \in \mathcal{O}(g(x))$, entonces $f(x)$ es del orden de $g(x)$).

Así, como vimos que las constantes no son relevantes para el análisis de la cota $\mathcal{O}$, existen ciertos *órdenes de complejidad* comunes para los programas que **no** varían por constantes. Reciben ese nombre porque justamente describen mediante notación $\mathcal{O}$ la complejidad de un programa.

Decidite con los nombres, ¿cota o *complejidad*?

- Antes mencionamos al pasar que la complejidad de un programa, normalmente llamado **complejidad computacional**, es la cantidad de recursos utilizados para correr un programa.
- Llamamos **complejidad asintótica computacional** al uso de *análisis asintótico* (como determinar la cota asintótica $\mathcal{O}$ con la que corre un programa) para estimar la complejidad computacional de un programa.
- Otra forma de decir que un programa "corre en $\mathcal{O}(n)$" es decir que es del **orden** $\mathcal{O}(n)$ (formalmente, si $f \in \mathcal{O}(g(x))$, entonces $f(x)$ es del orden de $g(x)$).

Así, como vimos que las constantes no son relevantes para el análisis de la cota $\mathcal{O}$, existen ciertos *órdenes de complejidad* comunes para los programas que **no** varían por constantes. Reciben ese nombre porque justamente describen mediante notación $\mathcal{O}$ la complejidad de un programa. Sirven como categorías para decir "en qué corren" los programas.

$\mathcal{O}(1)$

Decimos que un programa corre en tiempo **constante** o simplemente en "O en 1", y lo notamos $\mathcal{O}(1)$, si, como dice el nombre, su tiempo de ejecución está dado meramente por un factor constante, que no depende del tamaño de la entrada.

$\mathcal{O}(1)$

Decimos que un programa corre en tiempo **constante** o simplemente en "O en 1", y lo notamos $\mathcal{O}(1)$, si, como dice el nombre, su tiempo de ejecución está dado meramente por un factor constante, que no depende del tamaño de la entrada.

En Programación Competitiva los problemas cuya solución tiene esta cota asintótica suelen estar dados por cuentas matemáticas, por ejemplo.

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon)

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, **CodeForces 4A**)

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, **CodeForces 4A**)

↑

(Juez Online, más adelante)

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;
if ((w%2) == 0)
    cout << "YES";
else
    cout << "NO";
```

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;          ← 1 operación (leer un número)
if ((w%2) == 0)
    cout << "YES";
else
    cout << "NO";
```

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;          ← 1 operación (leer un número)
if ((w%2) == 0)           ← 2 operaciones (sacar resto %2 y comparar con 0)
    cout << "YES";
else
    cout << "NO";
```

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;          ← 1 operación (leer un número)
if ((w%2) == 0)           ← 2 operaciones (sacar resto %2 y comparar con 0)
    cout << "YES";        ← 1 operación (imprimir la respuesta)
else
    cout << "NO";         ← 1 operación (imprimir la respuesta)
```

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;        ← 1 operación (leer un número)
if ((w%2) == 0)         ← 2 operaciones (sacar resto %2 y comparar con 0)
    cout << "YES";      ← 1 operación (imprimir la respuesta)
else
    cout << "NO";       ← 1 operación (imprimir la respuesta)
```

Todas constantes, corre en $\mathcal{O}(1)$!

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;          ← 1 operación (leer un número)
if ((w%2) == 0)           ← 2 operaciones (sacar resto %2 y comparar con 0)
    cout << "YES";        ← 1 operación (imprimir la respuesta)
else
    cout << "NO";         ← 1 operación (imprimir la respuesta)
```

Todas constantes, corre en $\mathcal{O}(1)$! Hmmm, pero estará bien?

$\mathcal{O}(1)$ (ejemplo)

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso **par**. Dado w, ¿es posible?

En otras palabras, queremos determinar si existen dos números a y b tal que $a + b = w$ y a, b pares.
La suma de dos números pares siempre es par, entonces w **debe** ser par.

```
int w; cin >> w;          ← 1 operación (leer un número)
if ((w%2) == 0)           ← 2 operaciones (sacar resto %2 y comparar con 0)
    cout << "YES";        ← 1 operación (imprimir la respuesta)
else
    cout << "NO";         ← 1 operación (imprimir la respuesta)
```

Todas constantes, corre en $\mathcal{O}(1)$! Hmmm, pero estará bien?
Más adelante lo probaremos en el **juez**...

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de *n* elementos.

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de *n* elementos.

Observación

La base del logaritmo **no importa**.

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de *n* elementos.

Observación

La base del logaritmo **no importa**. En efecto, $log_a\, n$ y $log_b\, n$ para $a \neq b$ difieren en una *constante*.

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de *n* elementos.

Observación

La base del logaritmo **no importa**. En efecto, $log_a\, n$ y $log_b\, n$ para $a \neq b$ difieren en una *constante*. Por propiedad del logaritmo:

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de n elementos.

Observación

La base del logaritmo **no importa**. En efecto, $log_a\, n$ y $log_b\, n$ para $a \neq b$ difieren en una *constante*. Por propiedad del logaritmo:

$$\frac{log_a\, n}{log_a\, b} = log_b\, n \Rightarrow$$

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de n elementos.

Observación

La base del logaritmo **no importa**. En efecto, $log_a\, n$ y $log_b\, n$ para $a \neq b$ difieren en una *constante*. Por propiedad del logaritmo:

$$\frac{log_a\, n}{log_a\, b} = log_b\, n \Rightarrow log_a\, n = \quad log_a\, b \quad . \quad log_b\, n$$

$\mathcal{O}(log\ n)$

En este caso, decimos que un programa corre en tiempo **logarítmico** o en "O log n" (se lee tal cual) si el programa ejecuta *log n* operaciones, dada una entrada de n elementos.

Observación

La base del logaritmo **no importa**. En efecto, $log_a\, n$ y $log_b\, n$ para $a \neq b$ difieren en una *constante*. Por propiedad del logaritmo:

$$\frac{log_a\, n}{log_a\, b} = log_b\, n \Rightarrow log_a\, n = \underbrace{log_a\, b}_{constante} \ .\ log_b\, n$$

$\mathcal{O}(log\ n)$ (¿ejemplo?)

Si nos ponemos a pensar, no es tan fácil que una solución completa a un problema en Programación Competitiva tenga un orden de complejidad $\mathcal{O}(log\ n)$.

$\mathcal{O}(log\ n)$ (¿ejemplo?)

Si nos ponemos a pensar, no es tan fácil que una solución completa a un problema en Programación Competitiva tenga un orden de complejidad $\mathcal{O}(log\ n)$.

Al fin y al cabo, si tenemos que leer n elementos en la entrada, eso ya nos cuenta n operaciones, y como $n \leq log\ n$ para $n \geq 0$, el programa correrá en al menos $\mathcal{O}(n)$.

$\mathcal{O}(log\ n)$ (¿ejemplo?)

Si nos ponemos a pensar, no es tan fácil que una solución completa a un problema en Programación Competitiva tenga un orden de complejidad $\mathcal{O}(log\ n)$.
Al fin y al cabo, si tenemos que leer n elementos en la entrada, eso ya nos cuenta n operaciones, y como $n \leq log\ n$ para $n \geq 0$, el programa correrá en al menos $\mathcal{O}(n)$. Incluso si el programa que tenemos que hacer no lee los datos (por ejemplo en OIA), seguramente debamos recorrer todos los datos alguna vez para hacer algo con ellos...

$\mathcal{O}(log\ n)$ (¿ejemplo?)

Si nos ponemos a pensar, no es tan fácil que una solución completa a un problema en Programación Competitiva tenga un orden de complejidad $\mathcal{O}(log\ n)$.
Al fin y al cabo, si tenemos que leer n elementos en la entrada, eso ya nos cuenta n operaciones, y como $n \leq log\ n$ para $n \geq 0$, el programa correrá en al menos $\mathcal{O}(n)$. Incluso si el programa que tenemos que hacer no lee los datos (por ejemplo en OIA), seguramente debamos recorrer todos los datos alguna vez para hacer algo con ellos................

$\mathcal{O}(log\ n)$ (¿ejemplo?)

Si nos ponemos a pensar, no es tan fácil que una solución completa a un problema en Programación Competitiva tenga un orden de complejidad $\mathcal{O}(log\ n)$.
Al fin y al cabo, si tenemos que leer n elementos en la entrada, eso ya nos cuenta n operaciones, y como $n \leq log\ n$ para $n \geq 0$, el programa correrá en al menos $\mathcal{O}(n)$. Incluso si el programa que tenemos que hacer no lee los datos (por ejemplo en OIA), seguramente debamos recorrer todos los datos alguna vez para hacer algo con ellos................

Analicemos entonces la complejidad de un fragmento de código!

$\mathcal{O}(log\ n)$ (¡ejemplo!)

Problema (búsqueda binaria)

$\mathcal{O}(log\ n)$ (¡ejemplo!)

Problema (búsqueda binaria)

Dado un arreglo **ordenado** de n números enteros, queremos buscar el menor número de la lista que sea mayor o igual a un número x.

$\mathcal{O}(log\ n)$ (¡ejemplo!)

Problema (búsqueda binaria)

Dado un arreglo **ordenado** de n números enteros, queremos buscar el menor número de la lista que sea mayor o igual a un número x.

Este algoritmo es muy conocido. La solución consiste en aprovechar que el arreglo se encuentra ordenado para descartar la mitad de los números del arreglo en cada paso.

$\mathcal{O}(log\ n)$ (¡ejemplo!)

Problema (búsqueda binaria)

Dado un arreglo **ordenado** de n números enteros, queremos buscar el menor número de la lista que sea mayor o igual a un número x.

Este algoritmo es muy conocido. La solución consiste en aprovechar que el arreglo se encuentra ordenado para descartar la mitad de los números del arreglo en cada paso.

- Si la mediana del arreglo (`arr[n/2]`) es mayor o igual al número x, como los elementos en posiciones más grandes son todos mayores, sabemos que el número buscado está en en el intervalo $[0, \lfloor \frac{n}{2} \rfloor]$, descartando los últimos $\lceil \frac{n}{2} \rceil - 1$ elementos.

$\mathcal{O}(log\ n)$ (¡ejemplo!)

Problema (búsqueda binaria)

Dado un arreglo **ordenado** de n números enteros, queremos buscar el menor número de la lista que sea mayor o igual a un número x.

Este algoritmo es muy conocido. La solución consiste en aprovechar que el arreglo se encuentra ordenado para descartar la mitad de los números del arreglo en cada paso.

- Si la mediana del arreglo (`arr[n/2]`) es mayor o igual al número x, como los elementos en posiciones más grandes son todos mayores, sabemos que el número buscado está en en el intervalo $[0, \lfloor \frac{n}{2} \rfloor]$, descartando los últimos $\lceil \frac{n}{2} \rceil - 1$ elementos.
- Análogamente, si su mediana es menor a x, como todos los elementos en posiciones más chicas son también menores a x, sabemos que el número buscado está en el intervalo $[\lfloor \frac{n}{2} \rfloor, n-1]$, descartando los primeros $\lceil \frac{n}{2} \rceil$ elementos.

```cpp
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

```cpp
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {             ← Cada iteración hi-lo se reduce aprox. a la mitad
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

```
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

← Cada iteración `hi-lo` se reduce aprox. a la mitad (def. logaritmo, $\approx log_2\, n$ iteraciones)

```
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

← Cada iteración `hi-lo` se reduce aprox. a la mitad (def. logaritmo, $\approx log_2\, n$ iteraciones)

constante

```
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

← Cada iteración `hi-lo` se reduce aprox. a la mitad (def. logaritmo, $\approx log_2\, n$ iteraciones)

constante

De esta forma, corre en $\mathcal{O}(log\ n)$! (recordemos que la base del logaritmo no importaba)

```cpp
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {
    int mid = (hi+lo)/2;
    if (arr[mid] >= x) hi = mid;
    else lo = mid;
}
// El menor número mayor o igual a x está en arr[hi]
```

← Cada iteración `hi-lo` se reduce aprox. a la mitad (def. logaritmo, $\approx log_2\, n$ iteraciones)

} constante

De esta forma, corre en $\mathcal{O}(log\ n)$! (recordemos que la base del logaritmo no importaba)

En muchas soluciones de Prog. Competitiva se usa binary search **como parte de la solución**. Incluso es parte de la librería estándar de C++ (ver `std::lower_bound`).

```
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n-1;
while (hi-lo > 1) {                 ← Cada iteración hi-lo se reduce aprox. a la mitad
    int mid = (hi+lo)/2;               (def. logaritmo, ≈ log2 n iteraciones)
    if (arr[mid] >= x) hi = mid;
    else lo = mid;                  } constante
}
// El menor número mayor o igual a x está en arr[hi]
```

De esta forma, corre en $\mathcal{O}(log\ n)$! (recordemos que la base del logaritmo no importaba)

En muchas soluciones de Prog. Competitiva se usa binary search **como parte de la solución**. Incluso es parte de la librería estándar de C++ (ver `std::lower_bound`).
Es por esto que también es importante saber analizar la complejidad de "pedazos de programas", para luego obtener la cota asintótica $\mathcal{O}$ para la solución completa.

$\mathcal{O}(n)$

Un programa corre en tiempo **lineal** o en “O en n” si el programa ejecuta n operaciones, dados n elementos por entrada.

$\mathcal{O}(n)$

Un programa corre en tiempo **lineal** o en "O en n" si el programa ejecuta n operaciones, dados n elementos por entrada.

Ya vimos un ejemplo de un programa que corre en $\mathcal{O}(n)$ cuando definimos la notación $\mathcal{O}$: sumar los números de una lista.

$\mathcal{O}(n)$

Un programa corre en tiempo **lineal** o en "O en n" si el programa ejecuta n operaciones, dados n elementos por entrada.

Ya vimos un ejemplo de un programa que corre en $\mathcal{O}(n)$ cuando definimos la notación $\mathcal{O}$: sumar los números de una lista.
De igual forma, para mantener la costumbre, vamos a introducir otro ejemplo que nos será útil más adelante.

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión)

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, **Certamen Escolar 2021**)

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, **omegaUp 13769**)

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, omegaUp 13769)

$\uparrow$
(Juez Online)

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, omegaUp 13769)

Nicolás quiere que le compren N juguetes pero su mamá sólo le comprará $N - 1$. El chico le asignó un nivel de diversión positivo a cada uno de los N juguetes, dados por entrada. Queremos obtener la mayor diversión posible con $N - 1$ juguetes de los N dados.

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, omegaUp 13769)

Nicolás quiere que le compren N juguetes pero su mamá sólo le comprará $N - 1$. El chico le asignó un nivel de diversión positivo a cada uno de los N juguetes, dados por entrada. Queremos obtener la mayor diversión posible con $N - 1$ juguetes de los N dados.

Es decir, queremos obtener la mayor suma posible de $n - 1$ elementos de un arreglo de n enteros positivos.

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, omegaUp 13769)

Nicolás quiere que le compren N juguetes pero su mamá sólo le comprará $N-1$. El chico le asignó un nivel de diversión positivo a cada uno de los N juguetes, dados por entrada. Queremos obtener la mayor diversión posible con $N-1$ juguetes de los N dados.

Es decir, queremos obtener la mayor suma posible de $n-1$ elementos de un arreglo de n enteros positivos. Esto es equivalente a sumar todos los números y restar el de menor valor.

$\mathcal{O}(n)$ (otro ejemplo)

Problema (En busca de la mayor diversión, omegaUp 13769)

Nicolás quiere que le compren N juguetes pero su mamá sólo le comprará $N-1$. El chico le asignó un nivel de diversión positivo a cada uno de los N juguetes, dados por entrada. Queremos obtener la mayor diversión posible con $N-1$ juguetes de los N dados.

Es decir, queremos obtener la mayor suma posible de $n-1$ elementos de un arreglo de n enteros positivos. Esto es equivalente a sumar todos los números y restar el de menor valor.
El código debería ser sencillo, no?

```
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];
```

```cpp
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];

int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++) // Obtenemos su suma y el mínimo entre ellos
    sum += arr[i], mini = min(mini, arr[i]);
```

```cpp
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];

int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++) // Obtenemos su suma y el mínimo entre ellos
    sum += arr[i], mini = min(mini, arr[i]);

cout << sum - mini << '\n'; // Imprimimos la suma restado el nro. más chico
```

```cpp
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];

int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++) // Obtenemos su suma y el mínimo entre ellos
    sum += arr[i], mini = min(mini, arr[i]);

cout << sum - mini << '\n'; // Imprimimos la suma restado el nro. más chico
```

Rápidamente podemos ver que corre en $\mathcal{O}(n)$, pues cada `for` hace n iteraciones y ejecuta operaciones que suman a una constante.

```cpp
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];

int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++) // Obtenemos su suma y el mínimo entre ellos
    sum += arr[i], mini = min(mini, arr[i]);

cout << sum - mini << '\n'; // Imprimimos la suma restado el nro. más chico
```

Rápidamente podemos ver que corre en $\mathcal{O}(n)$, pues cada `for` hace n iteraciones y ejecuta operaciones que suman a una constante.

Pero el problema en sí tiene trampa...

```cpp
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leeemos los números
    cin >> arr[i];

int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++) // Obtenemos su suma y el mínimo entre ellos
    sum += arr[i], mini = min(mini, arr[i]);

cout << sum - mini << '\n'; // Imprimimos la suma restado el nro. más chico
```

Rápidamente podemos ver que corre en $\mathcal{O}(n)$, pues cada `for` hace n iteraciones y ejecuta operaciones que suman a una constante.

Pero el problema en sí tiene trampa... ya lo veremos cuando lo probemos en el **juez**.

$\mathcal{O}(n \log n)$

Entendemos que un programa corre en "O **n log n**" (hay mejores nombres como decir "linearítmico", pero no los usamos mucho) si su cota superior asintótica es $\mathcal{O}(n \log n)$. Es decir, si ejecuta alrededor de $n \log n$ operaciones dada una entrada de n elementos.

$\mathcal{O}(n \log n)$

Entendemos que un programa corre en "O **n log n**" (hay mejores nombres como decir "linearítmico", pero no los usamos mucho) si su cota superior asintótica es $\mathcal{O}(n \log n)$. Es decir, si ejecuta alrededor de $n \log n$ operaciones dada una entrada de n elementos.

Aunque no lo parezca, ya vimos también un ejemplo de un programa que corre en $\mathcal{O}(n \log n)$!

$\mathcal{O}(n \, log \, n)$

Entendemos que un programa corre en "O **n log n**" (hay mejores nombres como decir "linearítmico", pero no los usamos mucho) si su cota superior asintótica es $\mathcal{O}(n \, log \, n)$. Es decir, si ejecuta alrededor de $n \, log \, n$ operaciones dada una entrada de n elementos.

Aunque no lo parezca, ya vimos también un ejemplo de un programa que corre en $\mathcal{O}(n \, log \, n)$! Sí, el algoritmo de ordenamiento *merge sort* que antes pasamos por arriba tiene un orden de complejidad $n \, log \, n$.

$\mathcal{O}(n \log n)$

Entendemos que un programa corre en "O **n log n**" (hay mejores nombres como decir "linearítmico", pero no los usamos mucho) si su cota superior asintótica es $\mathcal{O}(n \log n)$. Es decir, si ejecuta alrededor de $n \log n$ operaciones dada una entrada de n elementos.

Aunque no lo parezca, ya vimos también un ejemplo de un programa que corre en $\mathcal{O}(n \log n)$! Sí, el algoritmo de ordenamiento *merge sort* que antes pasamos por arriba tiene un orden de complejidad $n \log n$.
Analicemos lo que hace para entender su complejidad.

$\mathcal{O}(n \log n)$ (ejemplo)

La idea fundamental de *merge sort* es la siguiente: si las dos mitades del arreglo divididas por un elemento central `arr[n/2]` estuvieran ordenadas, existe una forma eficiente de combinarlos para obtener un único arreglo ordenado?

$\mathcal{O}(n \log n)$ (ejemplo)

La idea fundamental de *merge sort* es la siguiente: si las dos mitades del arreglo divididas por un elemento central `arr[n/2]` estuvieran ordenadas, existe una forma eficiente de combinarlos para obtener un único arreglo ordenado? La respuesta como es de esperarse, es que sí!

$\mathcal{O}(n\ log\ n)$ (ejemplo)

La idea fundamental de *merge sort* es la siguiente: si las dos mitades del arreglo divididas por un elemento central `arr[n/2]` estuvieran ordenadas, existe una forma eficiente de combinarlos para obtener un único arreglo ordenado? La respuesta como es de esperarse, es que sí!

Esto es lo que hace la función `merge` (la más larga):

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.
   - Si vale, entonces el siguiente elemento de `arr` debe ser `a[aidx]`.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.
   - Si vale, entonces el siguiente elemento de `arr` debe ser `a[aidx]`.
   - Sino, debe ser `b[bidx]`.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.
   - Si vale, entonces el siguiente elemento de `arr` debe ser `a[aidx]`.
   - Sino, debe ser `b[bidx]`.

   En cada caso, se asigna el valor elegido a `arr[idx]`, avanzándose `idx`, y `aidx` o `bidx` según corresponda.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.
   - Si vale, entonces el siguiente elemento de `arr` debe ser `a[aidx]`.
   - Sino, debe ser `b[bidx]`.

   En cada caso, se asigna el valor elegido a `arr[idx]`, avanzándose `idx`, y `aidx` o `bidx` según corresponda.
3. Se repite el paso anterior hasta que alguno de los arreglos no tenga más elementos.

Sean ambas mitades los arreglos `a` y `b`, y el arreglo ordenado final formado por las dos mitades `arr`.

1. Se recorren ambas mitades al mismo tiempo desde el principio, manteniendo dos índices `aidx` y `bidx`.
   Además, se mantiene un índice `idx`, el próximo elemento a escribir en `arr` (empieza al principio).
2. En cada paso se compara `a[aidx] <= b[bidx]`.
   - Si vale, entonces el siguiente elemento de `arr` debe ser `a[aidx]`.
   - Sino, debe ser `b[bidx]`.

   En cada caso, se asigna el valor elegido a `arr[idx]`, avanzándose `idx`, y `aidx` o `bidx` según corresponda.
3. Se repite el paso anterior hasta que alguno de los arreglos no tenga más elementos.
4. Finalmente, se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay) en `arr`.

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una.

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares `a` y `b`.

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares `a` y `b`.

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares `a` y `b`.

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

$\boxed{\textbf{\textit{asz}} + \textbf{\textit{bsz}} \text{ operaciones}}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b.

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

$\boxed{\boldsymbol{n} \text{ operaciones}}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b.

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

$\boxed{\mathcal{O}(n)}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos.

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos.

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos.

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
```

$\boxed{\text{a lo sumo } \boldsymbol{asz} + \boldsymbol{bsz} \text{ iteraciones}}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos.

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
```

$\boxed{\text{a lo sumo } \boldsymbol{n} \text{ iteraciones}}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos.

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }
```

$\boxed{\mathcal{O}(n)}$

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay).

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay).

```
while (aidx < asz) {
    arr[idx] = a[aidx];
    idx++, aidx++;
}
```

```
while (bidx < bsz) {
    arr[idx] = b[bidx];
    idx++, bidx++;
}
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay).

```
while (aidx < asz) {
    arr[idx] = a[aidx];
    idx++, aidx++;
}
```

```
while (bidx < bsz) {
    arr[idx] = b[bidx];
    idx++, bidx++;
}
```

a lo sumo $\boldsymbol{max\{asz, bsz\}}$ iteraciones

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay).

```
while (aidx < asz) {
    arr[idx] = a[aidx];
    idx++, aidx++;
}
```

```
while (bidx < bsz) {
    arr[idx] = b[bidx];
    idx++, bidx++;
}
```

a lo sumo ***n*** iteraciones

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay). $\boxed{\mathcal{O}(n)}$

```
while (aidx < asz) {
    arr[idx] = a[aidx];
    idx++, aidx++;
}
```

```
while (bidx < bsz) {
    arr[idx] = b[bidx];
    idx++, bidx++;
}
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares `a` y `b`. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
```

```
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay). $\boxed{\mathcal{O}(n)}$

```
while (aidx < asz) arr[idx++] = a[aidx++];
while (bidx < bsz) arr[idx++] = b[bidx++];
```

De esta forma, se eligen los elementos en orden de ambas mitades, eligiendo en cada paso el elemento más chico al comienzo de cada una. En código:

```
void merge(vector<int> &arr, int l, int mid, int r) {
```

1. Se copian las mitades a arreglos auxiliares a y b. $\boxed{\mathcal{O}(n)}$

```
int asz = mid-l, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[l+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];
```

2. Se elige el primer elemento más pequeño de cada mitad mientras ambos tengan elementos. $\boxed{\mathcal{O}(n)}$

```
int aidx = 0, bidx = 0, idx = l;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];
```

3. Se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay). $\boxed{\mathcal{O}(n)}$

```
while (aidx < asz) arr[idx++] = a[aidx++];
while (bidx < bsz) arr[idx++] = b[bidx++];
```

```
}
```

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas...

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado.

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}
```

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}
```

← caso trivial (un sólo elemento)

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;

    int mid = (r+l)/2;
    split(arr,l,mid);
    split(arr,mid,r);
    merge(arr,l,mid,r);
}
```

← caso trivial (un sólo elemento)

} ordenar mitades *recursivamente*

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;            ← caso trivial (un sólo elemento)

    int mid = (r+l)/2;  }
    split(arr,l,mid);   }  ordenar mitades recursivamente
    split(arr,mid,r);   }
    merge(arr,l,mid,r); ← combinar mitades en un arreglo ordenado
}
```

Concluimos que `merge` corre en $\mathcal{O}(n)$, para un arreglo de n elementos.

Ahora bien, esto asume que ambas mitades del arreglo se encuentran ordenadas... entonces podemos correr `merge` de vuelta sobra cada una!

Repitiendo este proceso sucesivamente, eventualmente `merge` se llamará sobre un arreglo de un sólo elemento, donde su aplicación es trivial, porque ya se encuentra ordenado. Precisamente esto es lo que hace `split`:

```
void split(vector<int> &arr, int l, int r) {
    if (r-l <= 1)
        return;              ← caso trivial (un sólo elemento)

    int mid = (r+l)/2;
    split(arr,l,mid);        } ordenar mitades recursivamente
    split(arr,mid,r);
    merge(arr,l,mid,r);      ← combinar mitades en un arreglo ordenado
}
```

Notar que `split` ordena el intervalo $[l, r)$ del arreglo, pasándolo cada vez por referencia para evitar costos adicionales.

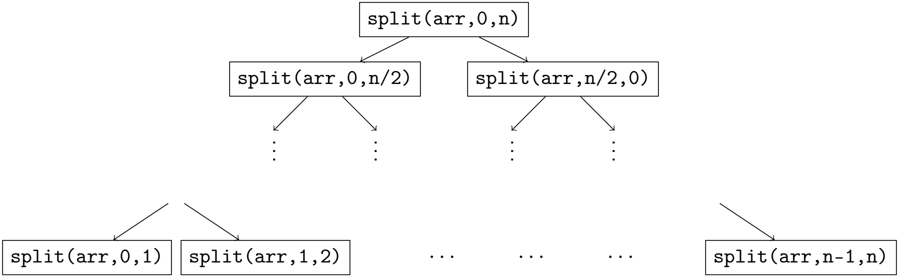
Finalmente, observemos que `split` hace a lo sumo $log_2 n$ llamadas recursivas en profundidad.

Finalmente, observemos que `split` hace a lo sumo $log_2 n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste.

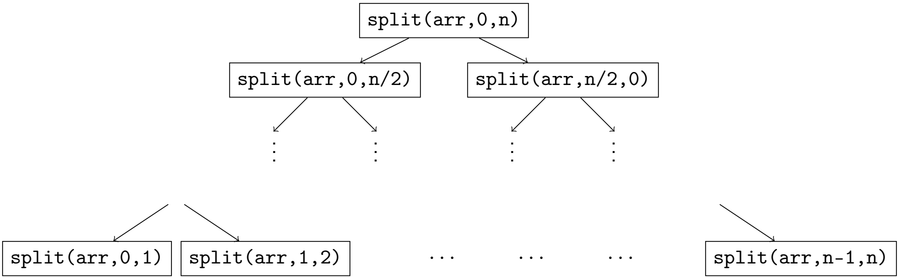
Finalmente, observemos que `split` hace a lo sumo $log_2\, n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2\, n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial.

Finalmente, observemos que `split` hace a lo sumo $log_2 n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2 n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

Finalmente, observemos que `split` hace a lo sumo $log_2\, n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2\, n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

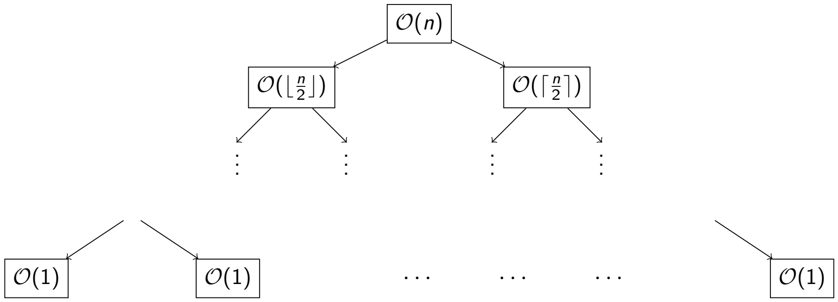


Finalmente, observemos que `split` hace a lo sumo $log_2\, n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2\, n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:



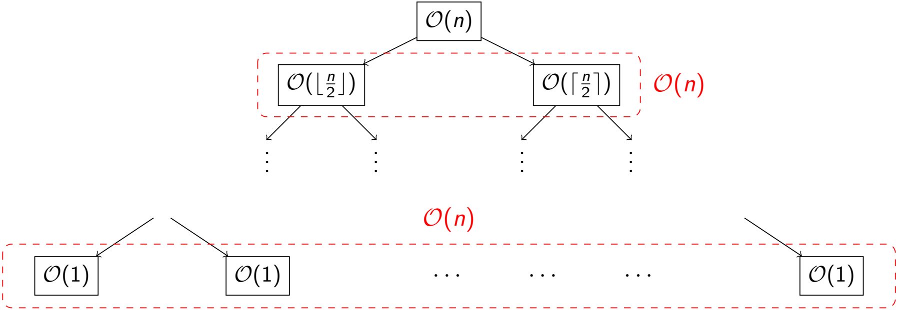
Recordando que cada llamada a `split` llama a `merge`, que tarda $\mathcal{O}(n)$...

Finalmente, observemos que `split` hace a lo sumo $log_2 n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2 n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:



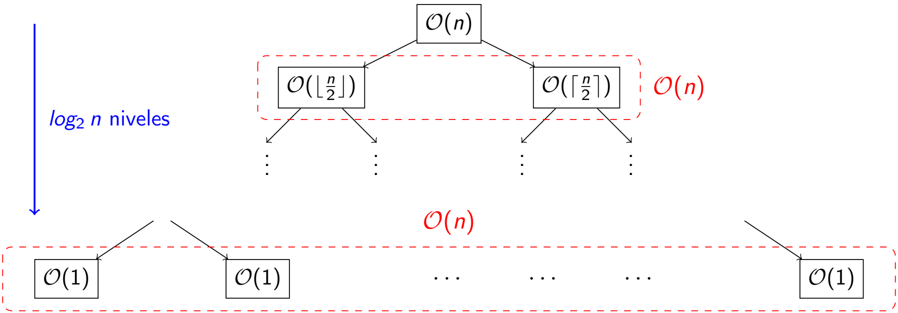
Recordando que cada llamada a `split` llama a `merge`, que tarda $\mathcal{O}(n)$...

Finalmente, observemos que `split` hace a lo sumo $log_2 n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2 n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

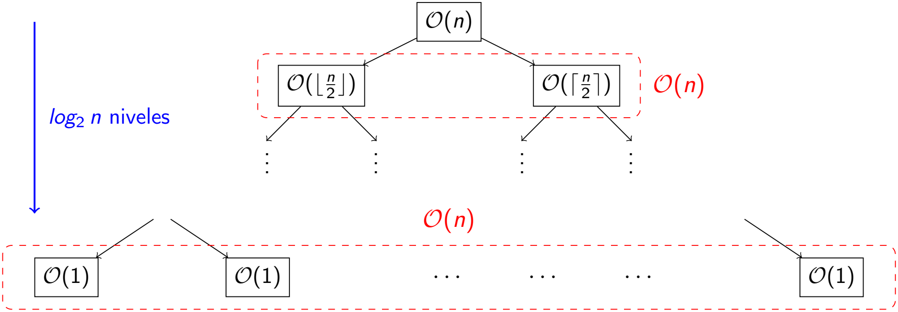


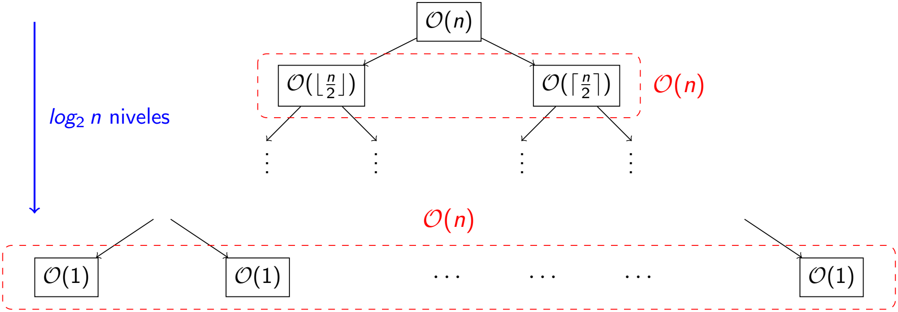
Recordando que cada llamada a `split` llama a `merge`, que tarda $\mathcal{O}(n)$...

Finalmente, observemos que `split` hace a lo sumo $log_2\, n$ llamadas recursivas en profundidad.
En efecto, en cada paso se llama `split` en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de $log_2\, n$ llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

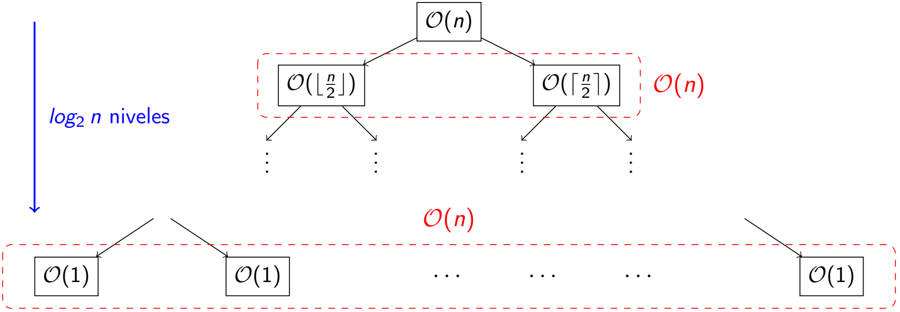


Recordando que cada llamada a `split` llama a `merge`, que tarda $\mathcal{O}(n)$...


$\mathcal{O}(n)$
$\mathcal{O}(\lfloor \frac{n}{2} \rfloor)$
$\mathcal{O}(\lceil \frac{n}{2} \rceil)$
$\mathcal{O}(n)$
$log_2\, n$ niveles
$\mathcal{O}(n)$
$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\cdots$ $\cdots$ $\cdots$
$\mathcal{O}(1)$

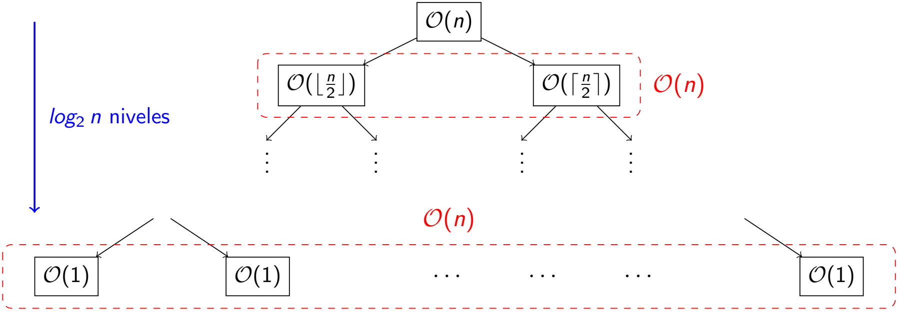


Así, como en cada nivel se hacen un total de operaciones del orden $\mathcal{O}(n)$, y la cadena de llamadas recursivas hecha por `split` alcanza a lo sumo una longitud de $log_2\, n$, concluimos que `split` corre en $\mathcal{O}(n\, log\, n)$ para un arreglo de n elementos en su intervalo $[0, n)$.



Así, como en cada nivel se hacen un total de operaciones del orden $\mathcal{O}(n)$, y la cadena de llamadas recursivas hecha por `split` alcanza a lo sumo una longitud de $log_2\, n$, concluimos que `split` corre en $\mathcal{O}(n\, log\, n)$ para un arreglo de n elementos en su intervalo $[0, n)$.

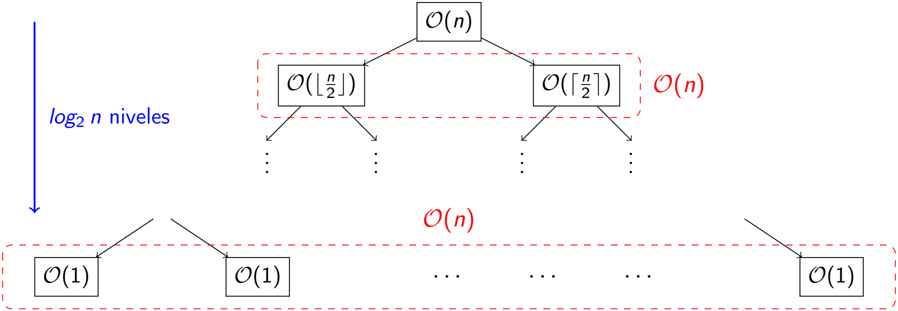
La última función que nos queda por analizar es `sort`, que simplemente llama a `split` de forma conveniente:



Así, como en cada nivel se hacen un total de operaciones del orden $\mathcal{O}(n)$, y la cadena de llamadas recursivas hecha por `split` alcanza a lo sumo una longitud de $log_2\ n$, concluimos que `split` corre en $\mathcal{O}(n\ log\ n)$ para un arreglo de n elementos en su intervalo $[0, n)$.

La última función que nos queda por analizar es `sort`, que simplemente llama a `split` de forma conveniente:

```
void sort(vector<int> &arr, int n) { split(arr,0,n); }
```



Así, como en cada nivel se hacen un total de operaciones del orden $\mathcal{O}(n)$, y la cadena de llamadas recursivas hecha por `split` alcanza a lo sumo una longitud de $log_2\, n$, concluimos que `split` corre en $\mathcal{O}(n\, log\, n)$ para un arreglo de n elementos en su intervalo $[0, n)$.

La última función que nos queda por analizar es `sort`, que simplemente llama a `split` de forma conveniente:

```
void sort(vector<int> &arr, int n) { split(arr,0,n); }
```

$\boxed{\mathcal{O}(n\, log\, n)}$

$\mathcal{O}(n^2)$

Un programa tiene un orden de complejidad **cuadrático**, o corre en "O n cuadrado", si dada una entrada de n elementos, ejecuta alrededor o a lo sumo n^2 operaciones.

$\mathcal{O}(n^2)$

Un programa tiene un orden de complejidad **cuadrático**, o corre en "O n cuadrado", si dada una entrada de n elementos, ejecuta alrededor o a lo sumo n^2 operaciones.

Si hacemos memoria al principio de la presentación, allí presentamos un algoritmo de ordenamiento que vimos que tardaba más tiempo en correr que merge sort sobre una lista de números dada.

$\mathcal{O}(n^2)$

Un programa tiene un orden de complejidad **cuadrático**, o corre en "O n cuadrado", si dada una entrada de n elementos, ejecuta alrededor o a lo sumo n^2 operaciones.

Si hacemos memoria al principio de la presentación, allí presentamos un algoritmo de ordenamiento que vimos que tardaba más tiempo en correr que merge sort sobre una lista de números dada.
En efecto, *selection sort* corre en $\mathcal{O}(n^2)$!

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]);
}
```

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]);
}
```

$\rightarrow$ n iteraciones

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]);
}
```

$\rightarrow$ n iteraciones

$\rightarrow$ $n - i - 1$ iteraciones

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[i])
                swap(arr[i],arr[j]);
}
```

$\rightarrow$ n iteraciones

$\rightarrow$ $n - i - 1$ iteraciones

$\left.\right\}$ constante

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)             → n iteraciones
        for (int j = i+1; j < n; j++)       → n − i − 1 iteraciones
            if (arr[j] < arr[i])            } constante
                swap(arr[i],arr[j]);        }
}
```

En este caso, al tener dos `for` anidados, la cantidad de operaciones que vamos a contar se multiplican.

$\mathcal{O}(n^2)$ (ejemplo)

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
    for (int i = 0; i < n; i++)              → n iteraciones
        for (int j = i+1; j < n; j++)        → n − i − 1 iteraciones
            if (arr[j] < arr[i])             } constante
                swap(arr[i],arr[j]);         }
}
```

En este caso, al tener dos `for` anidados, la cantidad de operaciones que vamos a contar se multiplican.
Para visualizar mejor la cantidad total de operaciones que se ejecutan, podemos representarla mediante una función y analizar bajo la definición formal.

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$f(n) = c(n-1) + c(n-2) + \cdots + 2c + c$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i)
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$\begin{aligned} f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\ &= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci \end{aligned}$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2}
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right)
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right) \leq c\,\frac{n^2}{2}
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right) \leq c\,\frac{n^2}{2} \leq cn^2
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right) \le c\,\frac{n^2}{2} \le cn^2 \\
&\boxed{\therefore f(n) \in \mathcal{O}(n^2)}
\end{aligned}
$$

Suponiendo que el costo constante en cada iteración es el mismo y está representado por c positivo, definimos la función:

$$
\begin{aligned}
f(n) &= c(n-1) + c(n-2) + \cdots + 2c + c \\
&= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c\sum_{i=1}^{n-1} i \\
&= c\,\frac{n(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right) \leq c\,\frac{n^2}{2} \leq cn^2 \\
&\boxed{\therefore f(n) \in \mathcal{O}(n^2)}
\end{aligned}
$$

Concluimos que el algoritmo corre en $\mathcal{O}(n^2)$.

Observación

Observación

- Si bien obtuvimos una cota asintótica $\mathcal{O}$ para el algoritmo mediante la definición formal, en la práctica no suele ser necesario.

Observación

- Si bien obtuvimos una cota asintótica $\mathcal{O}$ para el algoritmo mediante la definición formal, en la práctica no suele ser necesario.
- Basta observar que tenemos dos `for` anidados y la cantidad de iteraciones en cada uno como vimos al principio para aproximar que la función corre en $\mathcal{O}(n^2)$.

Observación

- Si bien obtuvimos una cota asintótica $\mathcal{O}$ para el algoritmo mediante la definición formal, en la práctica no suele ser necesario.
- Basta observar que tenemos dos `for` anidados y la cantidad de iteraciones en cada uno como vimos al principio para aproximar que la función corre en $\mathcal{O}(n^2)$.
- A pesar de esto, el análisis anterior resulta interesante para observar que incluso con el segundo `for` no necesariamente haciendo n iteraciones por cada una del primero, igualmente podemos decir que el orden de complejidad del algoritmo es $\mathcal{O}(n^2)$.

Otros órdenes de complejidad

El resto de órdenes de complejidad que uno se puede encontrar suele ser una combinación de los vistos, o bien de los siguientes, que valen la pena mencionar brevemente junto con algunos ejemplos de algoritmos conocidos:

Otros órdenes de complejidad

El resto de órdenes de complejidad que uno se puede encontrar suele ser una combinación de los vistos, o bien de los siguientes, que valen la pena mencionar brevemente junto con algunos ejemplos de algoritmos conocidos:

- $\mathcal{O}(\sqrt{n})$: Obtener la cantidad de divisores de un número / factorizar un número / determinar si un número es primo.
- $\mathcal{O}((n+q)\sqrt{n})$: Algoritmo de Mo (descomposición SQRT) (avanzado).
- $\mathcal{O}(n^c)$: Órdenes como $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$, etc. Típicamente suelen estar dados por varios `for` anidados. Ejemplos incluyen Floyd-Warshall, ciertas DPs o fuerzas brutas.
- $\mathcal{O}(2^n)$: Por ejemplo recorrer todas las máscaras de n bits, útil para enumerar posibles formas de tomar n elementos o DP Bitmask (avanzado).
- $\mathcal{O}(3^n)$: Iterar máscaras de n bits junto con sus submáscaras.
- $\mathcal{O}(n!)$: Recorrer todas las permutaciones de una lista de n elementos.

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

$$\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ log\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^c / c > 2) \subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!)$$

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

$$\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ log\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^c / c > 2)$$
$$\subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!)$$

donde pensamos a $\mathcal{O}(g(x))$ como el conjunto de las funciones f_i tal que $f_i \in \mathcal{O}(g(x))$.

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

$$\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\, log\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^c / c > 2)$$
$$\subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!)$$

donde pensamos a $\mathcal{O}(g(x))$ como el conjunto de las funciones f_i tal que $f_i \in \mathcal{O}(g(x))$.

De esta forma, vemos por ejemplo que toda función f_i que corre en $\mathcal{O}(1)$, por definición también tiene una cota asintótica $\mathcal{O}(n)$, pero no necesariamente al revés.

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

$$\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ log\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^c / c > 2) \\ \subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!)$$

donde pensamos a $\mathcal{O}(g(x))$ como el conjunto de las funciones f_i tal que $f_i \in \mathcal{O}(g(x))$.

De esta forma, vemos por ejemplo que toda función f_i que corre en $\mathcal{O}(1)$, por definición también tiene una cota asintótica $\mathcal{O}(n)$, pero no necesariamente al revés.
O sea, $f_i \in \mathcal{O}(1) \Rightarrow f_i \in \mathcal{O}(n)$, pero $f_i \in \mathcal{O}(n) \not\Rightarrow f_i \in \mathcal{O}(1)$.

De mejor a peor

Para cerrar la sección, podemos ver que los órdenes de complejidad mencionados se pueden "ordenar" en función de la cantidad de operaciones que representan respecto al tamaño de la entrada del programa:

$$\mathcal{O}(1) \subset \mathcal{O}(log\ n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ log\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^c / c > 2)$$
$$\subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(n!)$$

donde pensamos a $\mathcal{O}(g(x))$ como el conjunto de las funciones f_i tal que $f_i \in \mathcal{O}(g(x))$.

De esta forma, vemos por ejemplo que toda función f_i que corre en $\mathcal{O}(1)$, por definición también tiene una cota asintótica $\mathcal{O}(n)$, pero no necesariamente al revés.
O sea, $f_i \in \mathcal{O}(1) \Rightarrow f_i \in \mathcal{O}(n)$, pero $f_i \in \mathcal{O}(n) \not\Rightarrow f_i \in \mathcal{O}(1)$.

Por supuesto, **por conveniencia elegimos la cota más ajustada al describir la complejidad de un programa**.

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Como sugiere el título de la diapositiva, existe una rule of thumb bastante conocida en Programación Competitiva, para estimar el tiempo que tarda en ejecutarse una solución.

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Como sugiere el título de la diapositiva, existe una rule of thumb bastante conocida en Programación Competitiva, para estimar el tiempo que tarda en ejecutarse una solución.

Rule of thumb (operaciones en 1s)

Un programa que ejecuta alrededor de 10^8 operaciones corre en 1s.

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Como sugiere el título de la diapositiva, existe una rule of thumb bastante conocida en Programación Competitiva, para estimar el tiempo que tarda en ejecutarse una solución.

Rule of thumb (operaciones en 1s)

Un programa que ejecuta alrededor de 10^8 operaciones corre en 1s.

Dependiendo de la fuente este número puede variar, pudiendo llegar a argumentarse que hasta $3 \cdot 10^8$ o incluso $5 \cdot 10^8$ operaciones corren en 1 segundo.

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Como sugiere el título de la diapositiva, existe una rule of thumb bastante conocida en Programación Competitiva, para estimar el tiempo que tarda en ejecutarse una solución.

Rule of thumb (operaciones en 1s)

Un programa que ejecuta alrededor de 10^8 operaciones corre en 1s.

Dependiendo de la fuente este número puede variar, pudiendo llegar a argumentarse que hasta $3 \cdot 10^8$ o incluso $5 \cdot 10^8$ operaciones corren en 1 segundo.
Estas diferencias terminan reduciéndose al factor constante de la implementación, y no son muy relevantes considerando el objetivo principal de esta rule of thumb:

Rule of thumb de operaciones en 1 segundo

Una *rule of thumb* es una regla práctica, muchas veces aproximada, basada más en experiencia que en teoría.

Como sugiere el título de la diapositiva, existe una rule of thumb bastante conocida en Programación Competitiva, para estimar el tiempo que tarda en ejecutarse una solución.

Rule of thumb (operaciones en 1s)

Un programa que ejecuta alrededor de 10^8 operaciones corre en 1s.

Dependiendo de la fuente este número puede variar, pudiendo llegar a argumentarse que hasta $3 \cdot 10^8$ o incluso $5 \cdot 10^8$ operaciones corren en 1 segundo.
Estas diferencias terminan reduciéndose al factor constante de la implementación, y no son muy relevantes considerando el objetivo principal de esta rule of thumb: descartar soluciones "descabelladas" según el **tiempo límite** establecido para ejecución de la solución de un problema.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**:

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.
Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones. Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones. Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.
- **Merge sort**:

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones. Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.
- **Merge sort**: $\mathcal{O}(n \, log \, n)$.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones. Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.
- **Merge sort**: $\mathcal{O}(n \, log \, n)$. La base del logaritmo no importaba, así que tomemos log_2, para tomar un peor caso.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.
Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones.
  Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.
- **Merge sort**: $\mathcal{O}(n\, log\, n)$. La base del logaritmo no importaba, así que tomemos log_2, para tomar un peor caso.
  Ejecuta aproximadamente $10^5\, log_2\, 10^5 \approx 1{,}7 \cdot 10^6$ operaciones.

La liebre versus la tortuga: parte 2

Sabiendo la cota superior asintótica de merge sort y de selection sort, podemos aplicar la *rule of thumb* antes expuesta para entender mejor la diferencia de tiempo que observamos entre ambos algoritmos al principio de la presentación.

Recordemos que comparamos el tiempo que tomaba cada uno en ordenar una lista de $n = 10^5$ elementos:

- **Selection sort**: $\mathcal{O}(n^2)$. Corre $\approx (10^5)^2 = 10^{10}$ operaciones. Claramente $10^{10} \gg 10^8$, entonces está muy lejos de terminar en un segundo.
- **Merge sort**: $\mathcal{O}(n \, log \, n)$. La base del logaritmo no importaba, así que tomemos log_2, para tomar un peor caso.
  Ejecuta aproximadamente $10^5 \, log_2 \, 10^5 \approx 1{,}7 \cdot 10^6$ operaciones.
  Luego como $1{,}7 \cdot 10^6 < 10^8$, el algoritmo termina sin problemas en menos de 1s.

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números.

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no?

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.**

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.** Por dos motivos:

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.** Por dos motivos:

- El más obvio es que ya existe una función en la librería estándar de C++ para ordenar números: `std::sort`.

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.** Por dos motivos:

- El más obvio es que ya existe una función en la librería estándar de C++ para ordenar números: `std::sort`.
- El segundo motivo es el más importante para Programación Competitiva.

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.** Por dos motivos:

- El más obvio es que ya existe una función en la librería estándar de C++ para ordenar números: `std::sort`.
- El segundo motivo es el más importante para Programación Competitiva.
  Si estimamos que una solución que corre en $\mathcal{O}(n^2)$ es suficiente ejecutar la solución en el *tiempo límite* estipulado para un problema, pero es más corta y/o fácil de escribir que otra más eficiente, **preferimos la más rápida de escribir**.

La liebre versus la tortuga: final part

A este punto creo que dejamos en claro que merge sort es mucho mejor que selection sort como algoritmo de ordenamiento de números. Eso significa que siempre que necesitemos un algoritmo para ordenar números usamos merge sort, no? **NO.** Por dos motivos:

- El más obvio es que ya existe una función en la librería estándar de C++ para ordenar números: `std::sort`.
- El segundo motivo es el más importante para Programación Competitiva.
  Si estimamos que una solución que corre en $\mathcal{O}(n^2)$ es suficiente ejecutar la solución en el *tiempo límite* estipulado para un problema, pero es más corta y/o fácil de escribir que otra más eficiente, **preferimos la más rápida de escribir**.
  Un código más largo y/o complejo es más propenso a errores, sin mencionar que el tiempo de la competencia es limitado.

Algunas operaciones pesan más que otras

Una última observación que vale la pena hacer es que no todas las "operaciones" tardan lo mismo.

Algunas operaciones pesan más que otras

Una última observación que vale la pena hacer es que no todas las "operaciones" tardan lo mismo.

Cuando introdujimos el concepto de operaciones no definimos mucho qué consideramos como una operación.

Algunas operaciones pesan más que otras

Una última observación que vale la pena hacer es que no todas las "operaciones" tardan lo mismo.

Cuando introdujimos el concepto de operaciones no definimos mucho qué consideramos como una operación. Intuitivamente, lo pensamos como alguna comparación, alguna suma, una llamada a una función (sin contar las operaciones que hace dentro), leer o escribir un tipo primitivo (como un número), etc.

Algunas operaciones pesan más que otras

Una última observación que vale la pena hacer es que no todas las "operaciones" tardan lo mismo.

Cuando introdujimos el concepto de operaciones no definimos mucho qué consideramos como una operación. Intuitivamente, lo pensamos como alguna comparación, alguna suma, una llamada a una función (sin contar las operaciones que hace dentro), leer o escribir un tipo primitivo (como un número), etc.
Sin embargo, no todas las operaciones tardan lo mismo en ejecutarse. Una suma suele ser mucho más rápida que una multiplicación, que a su vez es más rápida que una división.

Algunas operaciones pesan más que otras

Una última observación que vale la pena hacer es que no todas las "operaciones" tardan lo mismo.

Cuando introdujimos el concepto de operaciones no definimos mucho qué consideramos como una operación. Intuitivamente, lo pensamos como alguna comparación, alguna suma, una llamada a una función (sin contar las operaciones que hace dentro), leer o escribir un tipo primitivo (como un número), etc.
Sin embargo, no todas las operaciones tardan lo mismo en ejecutarse. Una suma suele ser mucho más rápida que una multiplicación, que a su vez es más rápida que una división.

Este antecedente se suma a los motivos por las cuales solemos ignorar las constantes al analizar la complejidad de un programa.

Así, en los casos cuando es realmente necesario "ajustar el factor constante", se suelen reemplazar el uso de ciertas estructuras de la librería estándar `std::map` o `std::set` por búsquedas en arreglos ordenados siempre que sea posible, o evitar repetir operaciones sobre éstos haciendo uso de *iteradores*.

Así, en los casos cuando es realmente necesario "ajustar el factor constante", se suelen reemplazar el uso de ciertas estructuras de la librería estándar `std::map` o `std::set` por búsquedas en arreglos ordenados siempre que sea posible, o evitar repetir operaciones sobre éstos haciendo uso de *iteradores*.
Sin mencionar que muchas otras optimizaciones menores hoy en día las realiza el compilador automáticamente.

Así, en los casos cuando es realmente necesario "ajustar el factor constante", se suelen reemplazar el uso de ciertas estructuras de la librería estándar `std::map` o `std::set` por búsquedas en arreglos ordenados siempre que sea posible, o evitar repetir operaciones sobre éstos haciendo uso de *iteradores*.
Sin mencionar que muchas otras optimizaciones menores hoy en día las realiza el compilador automáticamente.

Como conclusión, es difícil "medir" la cantidad de operaciones, y más allá de los lineamentos generales vistos, en parte **se aprende con la práctica**.

Así, en los casos cuando es realmente necesario “ajustar el factor constante”, se suelen reemplazar el uso de ciertas estructuras de la librería estándar `std::map` o `std::set` por búsquedas en arreglos ordenados siempre que sea posible, o evitar repetir operaciones sobre éstos haciendo uso de *iteradores*.
Sin mencionar que muchas otras optimizaciones menores hoy en día las realiza el compilador automáticamente.

Como conclusión, es difícil “medir” la cantidad de operaciones, y más allá de los lineamentos generales vistos, en parte **se aprende con la práctica**.

Y reducir el factor constante de un programa no sólo muchas veces no es necesario, sino que en caso de serlo, se realiza en todo caso optimizando las operaciones “más pesadas”, conocimiento que también surge **de la experiencia**.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values)

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, **CSES 1645**)

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, **CSES 1645**)

$\uparrow$
(Juez Online)

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $\boldsymbol{1 \leq n \leq 2 \cdot 10^5}$

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera.
$1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera.
$1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera.
$1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \log n)$.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \, log \, n)$. $\approx 3{,}5 \cdot 10^6$ operaciones para $2 \cdot 10^5$ números.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \log n)$. $\approx 3{,}5 \cdot 10^6$ operaciones para $2 \cdot 10^5$ números. ✓, pero TREMENDO overkill.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera.
$1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \log n)$. $\approx 3{,}5 \cdot 10^6$ operaciones para $2 \cdot 10^5$ números. ✓, pero TREMENDO overkill.
- ???.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \, log \, n)$. $\approx 3{,}5 \cdot 10^6$ operaciones para $2 \cdot 10^5$ números. ✓, pero TREMENDO overkill.
- ???. $\mathcal{O}(n)$.

Nueva sección, nuevo problema

Problema (Nearest Smaller Values, CSES 1645)

Dado un arreglo de n enteros, encontrar para cada posición del arreglo la posición más cercana a su izquierda que tenga un menor valor a la primera. $1 \leq n \leq 2 \cdot 10^5$

¿Soluciones?

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. ✗, muy lento para $2 \cdot 10^5$ números.
- (avanzado) SQRT decomposition. $\mathcal{O}(n\sqrt{n})$. $\approx 7 \cdot 10^7$ operaciones para $2 \cdot 10^5$ números. ✓, pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \, log \, n)$. $\approx 3{,}5 \cdot 10^6$ operaciones para $2 \cdot 10^5$ números. ✓, pero TREMENDO overkill.
- ???. $\mathcal{O}(n)$. ✓, simple y corta.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.
   Por lo tanto, el primero menor a izq de `arr[i]` también puede ser el primero menor a izq de `arr[i-1]`, que ya lo calculamos.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.
   Por lo tanto, el primero menor a izq de `arr[i]` también puede ser el primero menor a izq de `arr[i-1]`, que ya lo calculamos.
    1. Si ese número es menor a `arr[i]`, obtuvimos la respuesta.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.
   Por lo tanto, el primero menor a izq de `arr[i]` también puede ser el primero menor a izq de `arr[i-1]`, que ya lo calculamos.
   1. Si ese número es menor a `arr[i]`, obtuvimos la respuesta.
   2. Sino, repetimos el razonamiento anterior para obtener el primero menor a izq del obtenido.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.

   Por lo tanto, el primero menor a izq de `arr[i]` también puede ser el primero menor a izq de `arr[i-1]`, que ya lo calculamos.
   1. Si ese número es menor a `arr[i]`, obtuvimos la respuesta.
   2. Sino, repetimos el razonamiento anterior para obtener el primero menor a izq del obtenido.

   Observar que no nos salteamos ningún candidato posible, ya que si hubiera existido, hubiera sido también el primero menor de `arr[i-1]`, pues es mayor o igual a `arr[i]`.

La solución ???

Supongamos que queremos obtener el índice del primer número menor a la izquierda de `arr[i]`, y ya lo calculamos para todos los números en posiciones más chicas que i. Veamos que:

1. Si `arr[i-1] < arr[i]`, la respuesta es `i-1`.
2. Sino, vale que `arr[i-1] >= arr[i]`.
   Por lo tanto, el primero menor a izq de `arr[i]` también puede ser el primero menor a izq de `arr[i-1]`, que ya lo calculamos.
   1. Si ese número es menor a `arr[i]`, obtuvimos la respuesta.
   2. Sino, repetimos el razonamiento anterior para obtener el primero menor a izq del obtenido.

   Observar que no nos salteamos ningún candidato posible, ya que si hubiera existido, hubiera sido también el primero menor de `arr[i-1]`, pues es mayor o igual a `arr[i]`.
3. Repitiendo el paso anterior sucesivamente, eventualmente obtendremos el primero menor de `arr[i]` (si existe), ya que saltamos por números de forma decreciente hacia índices más chicos.

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba `0` (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` $\leftarrow$ constante

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` $\leftarrow$ constante

3. Para cada número en las posiciones $[1, n]$...

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` ← constante

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` ← constante

3. Para cada número en las posiciones $[1, n]$...

`for (int i = 1; i <= n; i++) {` ← n iteraciones

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` $\leftarrow$ constante

3. Para cada número en las posiciones $[1, n]$...

`for (int i = 1; i <= n; i++) {` $\leftarrow$ n iteraciones

...obtenemos su primero menor a izq como describimos en la idea...

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;
```

← constante

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {
```

← n iteraciones

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;     ← constante
```

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {      ← n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                    ← constante
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;                     ← constante
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` $\leftarrow$ constante

3. Para cada número en las posiciones $[1, n]$...

`for (int i = 1; i <= n; i++) {` $\leftarrow$ n iteraciones

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                    ← constante
    while (arr[j] >= arr[i])        } ??? operaciones
        j = ans[j];                 }
    ans[i] = j;                     ← constante
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;      ← constante
```

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {      ← n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                    ← constante
    while (arr[j] >= arr[i])        } ??? operaciones
        j = ans[j];                 }
    ans[i] = j;                     ← constante
```

...y finalmente lo escribimos a la salida.

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;     ← constante
```

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {      ← n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                    ← constante
    while (arr[j] >= arr[i])        } ??? operaciones
        j = ans[j];                 }
    ans[i] = j;                     ← constante
```

...y finalmente lo escribimos a la salida.

```
        cout << j << ' ';
    }
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

```
arr[0] = -1;
```
← constante

3. Para cada número en las posiciones $[1, n]$...

```
for (int i = 1; i <= n; i++) {
```
← n iteraciones

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                  ← constante
    while (arr[j] >= arr[i])      } ??? operaciones
        j = ans[j];               }
    ans[i] = j;                   ← constante
```

...y finalmente lo escribimos a la salida.

```
        cout << j << ' ';         ← constante
    }
```

¿Cuál es ese Pokémon?

Analicemos la complejidad de la solución viendo su implementación:

1. Leemos la entrada en posiciones 1 a n.

```
int n; cin >> n;
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

$\boxed{\mathcal{O}(n)}$

2. Establecemos `arr[0]` en un número más chico al resto del arreglo, para que si no se encuentra primero menor, se encuentre `arr[0]` y se escriba 0 (el enunciado original establece que $1 \leq$ `arr[i]` $\leq 10^9$)...

`arr[0] = -1;` ← constante

3. Para cada número en las posiciones $[1, n]$...

`for (int i = 1; i <= n; i++) {` ← n iteraciones

...obtenemos su primero menor a izq como describimos en la idea...

```
    int j = i-1;                  ← constante
    while (arr[j] >= arr[i])      } ??? operaciones
        j = ans[j];
    ans[i] = j;                   ← constante
```

$\boxed{\mathcal{O}(n \cdot ???)}$

...y finalmente lo escribimos a la salida.

```
        cout << j << ' ';         ← constante
    }
```

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

A simple vista, podríamos decir que realiza a lo sumo i iteraciones, ya que `ans[j]` contiene el índice del primero menor al número en la posición j, así a lo sumo recorriendo todos los números anteriores a la posición i.

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

A simple vista, podríamos decir que realiza a lo sumo i iteraciones, ya que `ans[j]` contiene el índice del primero menor al número en la posición j, así a lo sumo recorriendo todos los números anteriores a la posición i.
Si nos basamos en esto, la complejidad de esta solución sería $\mathcal{O}(n^2)$, que dijimos es inaceptable para este problema.

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

A simple vista, podríamos decir que realiza a lo sumo i iteraciones, ya que `ans[j]` contiene el índice del primero menor al número en la posición j, así a lo sumo recorriendo todos los números anteriores a la posición i.
Si nos basamos en esto, la complejidad de esta solución sería $\mathcal{O}(n^2)$, que dijimos es inaceptable para este problema.

Para obtener una mejor cota asintótica superior, pensemos en una forma distinta de contar las operaciones dentro del último `for`:

Concentrémonos en el último `while`, para determinar la cantidad de iteraciones que realiza en cada una del `for`:

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

A simple vista, podríamos decir que realiza a lo sumo i iteraciones, ya que `ans[j]` contiene el índice del primero menor al número en la posición j, así a lo sumo recorriendo todos los números anteriores a la posición i.
Si nos basamos en esto, la complejidad de esta solución sería $\mathcal{O}(n^2)$, que dijimos es inaceptable para este problema.

Para obtener una mejor cota asintótica superior, pensemos en una forma distinta de contar las operaciones dentro del último `for`:

¿Cuántas operaciones se ejecutan por cada elemento del arreglo?

Podemos dividirlas en dos tipos:

Podemos dividirlas en dos tipos:

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).
2. Cuando lo comparamos con otro elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando `j` es el índice del elemento

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).
2. Cuando lo comparamos con otro elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando `j` es el índice del elemento (costo ???).

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).
2. Cuando lo comparamos con otro elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando `j` es el índice del elemento (costo ???).

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

En otras palabras, asignamos las operaciones dentro del `while` al elemento `arr[j]`, y las de fuera al elemento `arr[i]`.

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).
2. Cuando lo comparamos con otro elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando `j` es el índice del elemento (costo ???).

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

En otras palabras, asignamos las operaciones dentro del `while` al elemento `arr[j]`, y las de fuera al elemento `arr[i]`.

Observemos entonces que para cada elemento, sólo se hace una cantidad constante de operaciones del tipo 2!

Podemos dividirlas en dos tipos:

1. Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando `i` es el índice del elemento (costo constante).
2. Cuando lo comparamos con otro elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando `j` es el índice del elemento (costo ???).

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}
```

En otras palabras, asignamos las operaciones dentro del `while` al elemento `arr[j]`, y las de fuera al elemento `arr[i]`.

Observemos entonces que para cada elemento, sólo se hace una cantidad constante de operaciones del tipo 2!
En efecto, veamos que cada elemento `arr[j]` sólo se lo recorrerá una sóla vez con el `while` (**valiendo la condición**) entre todas las iteraciones del `for`:

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr[`i'`]` con $i' > i$, nos encontramos con dos casos:

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr[`i'`]` con $i' > i$, nos encontramos con dos casos:

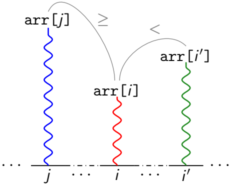
1. <u>`arr[`i'`]` $>$ `arr[`i`]`</u>:

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr[`i'`]` con $i' > i$, nos encontramos con dos casos:

1. `arr[`i'`]` $>$ `arr[`i`]`: El primero menor a izquierda de `arr[`i'`]` es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).

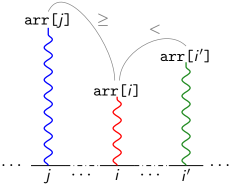
Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr`$[i']$ con $i' > i$, nos encontramos con dos casos:

1. `arr`$[i']$ $>$ `arr`$[i]$: El primero menor a izquierda de `arr`$[i']$ es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).



Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr`$[i']$ con $i' > i$, nos encontramos con dos casos:

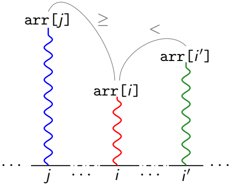
1. `arr`$[i']$ $>$ `arr`$[i]$: El primero menor a izquierda de `arr`$[i']$ es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).



2. `arr`$[i']$ $\leq$ `arr`$[i]$:

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de $\texttt{arr}[i']$ con $i' > i$, nos encontramos con dos casos:

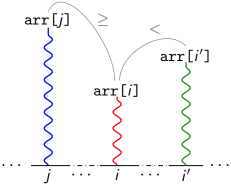
1. $\underline{\texttt{arr}[i'] > \texttt{arr}[i]}$: El primero menor a izquierda de $\texttt{arr}[i']$ es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).



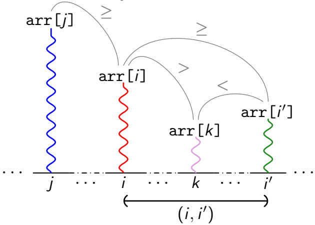
2. $\underline{\texttt{arr}[i'] \leq \texttt{arr}[i]}$: Si para algún índice k en el intervalo (i, i'), $\texttt{arr}[k] < \texttt{arr}[i']$, luego no se recorre en el `while` ningún índice menor, en particular **no** recorriéndose $\texttt{arr}[j]$ pues $j < i < k < i'$.

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr`$[i']$ con $i' > i$, nos encontramos con dos casos:

1. `arr`$[i']$ $>$ `arr`$[i]$: El primero menor a izquierda de `arr`$[i']$ es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).

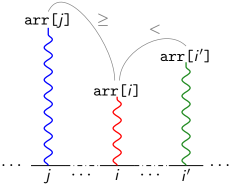


2. `arr`$[i']$ $\leq$ `arr`$[i]$: Si para algún índice k en el intervalo (i, i'), `arr`$[k]$ $<$ `arr`$[i']$, luego no se recorre en el `while` ningún índice menor, en particular **no** recorriéndose `arr`$[j]$ pues $j < i < k < i'$.

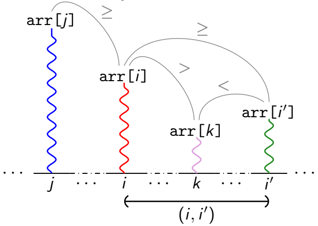


Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr[`i'`]` con $i' > i$, nos encontramos con dos casos:

1. <u>`arr[`i'`]` $>$ `arr[`i`]`</u>: El primero menor a izquierda de `arr[`i'`]` es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).



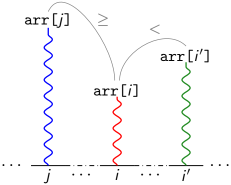
2. <u>`arr[`i'`]` $\leq$ `arr[`i`]`</u>: Si para algún índice k en el intervalo (i, i'), `arr[`k`]` $<$ `arr[`i'`]`, luego no se recorre en el `while` ningún índice menor, en particular **no** recorriéndose `arr[`j`]` pues $j < i < k < i'$.



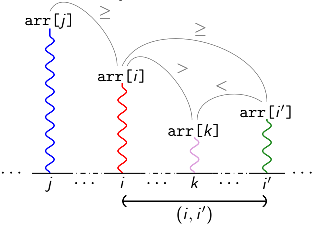
Sino, en el `while` se recorre a `arr[`i`]`, ya que debe ser primero menor de algún elemento en (i, i').

Si se recorre `arr[j]` **valiendo la condición** en una operación de tipo 2, es porque `arr[j]` $\geq$ `arr[i]` donde `i` se corresponde el índice del `for`.
Así, al buscar el primero menor a izquierda de `arr`$[i']$ con $i' > i$, nos encontramos con dos casos:

1. `arr`$[i']$ $>$ `arr`$[i]$: El primero menor a izquierda de `arr`$[i']$ es `arr[i]` o un número en un índice mayor a i, **evitándose** recorrer `arr[j]` en el `while` (recorre números de forma decreciente).



2. `arr`$[i']$ $\leq$ `arr`$[i]$: Si para algún índice k en el intervalo (i, i'), `arr`$[k]$ $<$ `arr`$[i']$, luego no se recorre en el `while` ningún índice menor, en particular **no** recorriéndose `arr`$[j]$ pues $j < i < k < i'$.



Sino, en el `while` se recorre a `arr`$[i]$, ya que debe ser primero menor de algún elemento en (i, i').

Luego, nuevamente, como `arr`$[i]$ $\leq$ `arr`$[j]$, **no** se recorre a `arr`$[j]$.

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.
- Por otro lado, si se lo itera en el `while` **sin** valer la condición (en la última iteración del loop), se puede contar como una operación más para cada elemento del arreglo (respecto a `arr[`i`]`), ya que el `while` termina exactamente una vez por cada i del `for`.

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.
- Por otro lado, si se lo itera en el `while` **sin** valer la condición (en la última iteración del loop), se puede contar como una operación más para cada elemento del arreglo (respecto a `arr[`i`]`), ya que el `while` termina exactamente una vez por cada i del `for`.
  Es decir, contamos esta última operación como parte de "encontrar el primero menor a izquierda" de `arr[`i`]`, siendo de tipo 1, y mantiene el costo de este tipo de operaciones para cada elemento **constante** (+1 de las vistas antes).

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.
- Por otro lado, si se lo itera en el `while` **sin** valer la condición (en la última iteración del loop), se puede contar como una operación más para cada elemento del arreglo (respecto a arr[i]), ya que el `while` termina exactamente una vez por cada i del `for`.
  Es decir, contamos esta última operación como parte de "encontrar el primero menor a izquierda" de arr[i], siendo de tipo 1, y mantiene el costo de este tipo de operaciones para cada elemento **constante** (+1 de las vistas antes).
- Así, la cantidad de operaciones de tipo 2 por cada elemento también es **constante** (una sóla iteración del `while` para c/elemento).

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.
- Por otro lado, si se lo itera en el `while` **sin** valer la condición (en la última iteración del loop), se puede contar como una operación más para cada elemento del arreglo (respecto a arr[i]), ya que el `while` termina exactamente una vez por cada i del `for`.
  Es decir, contamos esta última operación como parte de "encontrar el primero menor a izquierda" de arr[i], siendo de tipo 1, y mantiene el costo de este tipo de operaciones para cada elemento **constante** (+1 de las vistas antes).
- Así, la cantidad de operaciones de tipo 2 por cada elemento también es **constante** (una sóla iteración del `while` para c/elemento).
- Finalmente, como se ejecuta una cantidad **constante** de operaciones por cada elemento en el `for` visto (y ninguna más), su complejidad es $\mathcal{O}(n)$.

- Probamos entonces que cada elemento del arreglo se lo itera por el `while` **valiendo la condición** a lo sumo una vez entre todas las iteraciones del `for`.
- Por otro lado, si se lo itera en el `while` **sin** valer la condición (en la última iteración del loop), se puede contar como una operación más para cada elemento del arreglo (respecto a `arr[`i`]`), ya que el `while` termina exactamente una vez por cada i del `for`.
  Es decir, contamos esta última operación como parte de "encontrar el primero menor a izquierda" de `arr[`i`]`, siendo de tipo 1, y mantiene el costo de este tipo de operaciones para cada elemento **constante** (+1 de las vistas antes).
- Así, la cantidad de operaciones de tipo 2 por cada elemento también es **constante** (una sóla iteración del `while` para c/elemento).
- Finalmente, como se ejecuta una cantidad **constante** de operaciones por cada elemento en el `for` visto (y ninguna más), su complejidad es $\mathcal{O}(n)$.
- Por lo tanto, **toda la solución corre en $\mathcal{O}(n)$**.

¿Analizar cota $\mathcal{O}$ no era fácil?

Varias veces, sí. Pero no siempre.

¿Analizar cota $\mathcal{O}$ no era fácil?

Varias veces, sí. Pero no siempre.

Hay casos como el que vimos, donde parte de resolver el problema es ver que una solución que a simple vista parece costosa, en realidad tiene una cota asintótica superior muy buena.

¿Analizar cota $\mathcal{O}$ no era fácil?

Varias veces, sí. Pero no siempre.

Hay casos como el que vimos, donde parte de resolver el problema es ver que una solución que a simple vista parece costosa, en realidad tiene una cota asintótica superior muy buena.

Un análisis como el anterior, donde estudiamos en detalle cómo se ejecutan las operaciones observando cuáles son posibles y cuáles no, recibe el nombre de **análisis amortizado**.

¿Analizar cota $\mathcal{O}$ no era fácil?

Varias veces, sí. Pero no siempre.

Hay casos como el que vimos, donde parte de resolver el problema es ver que una solución que a simple vista parece costosa, en realidad tiene una cota asintótica superior muy buena.

Un análisis como el anterior, donde estudiamos en detalle cómo se ejecutan las operaciones observando cuáles son posibles y cuáles no, recibe el nombre de **análisis amortizado**.
Éste es útil cuando la cota que obtendríamos observando "el peor caso", acotando de forma sencilla las operaciones que realizamos, puede ser demasiado *pesimista*, requiriendo así una observación más exhaustiva para obtener una cota superior más ajustada.

Miles de problemas para entrenar

Existen muchas páginas en internet con no sólo enunciados de problemas para practicar, sino también con la opción de **subir soluciones** para que sean evaluadas en el momento.

Miles de problemas para entrenar

Existen muchas páginas en internet con no sólo enunciados de problemas para practicar, sino también con la opción de **subir soluciones** para que sean evaluadas en el momento.
No se suben ejecutables, sino sólo código fuente que es compilado y ejecutado en el servidor contra un evaluador y casos de prueba.

Miles de problemas para entrenar

Existen muchas páginas en internet con no sólo enunciados de problemas para practicar, sino también con la opción de **subir soluciones** para que sean evaluadas en el momento.

No se suben ejecutables, sino sólo código fuente que es compilado y ejecutado en el servidor contra un evaluador y casos de prueba.

A estas páginas les llamamos **jueces online**, pues hacen de "jueces" evaluando nuestro código y determinando un veredicto: si la solución es correcta o incorrecta...

Miles de problemas para entrenar

Existen muchas páginas en internet con no sólo enunciados de problemas para practicar, sino también con la opción de **subir soluciones** para que sean evaluadas en el momento.
No se suben ejecutables, sino sólo código fuente que es compilado y ejecutado en el servidor contra un evaluador y casos de prueba.
A estas páginas les llamamos **jueces online**, pues hacen de "jueces" evaluando nuestro código y determinando un veredicto: si la solución es correcta o incorrecta... con ciertas variantes.

¿Qué es omegaUp?

¿Qué es omegaUp?

omegaUp es un juez online muy utilizado en México que permite fácilmente crear competencias y subir problemas nuevos con distintos formatos. Debido a su conveniencia, se lo utiliza para llevar a cabo el Certamen Escolar todos los años y el Torneo Jujeño de Programación.

¿Qué es omegaUp?

omegaUp es un juez online muy utilizado en México que permite fácilmente crear competencias y subir problemas nuevos con distintos formatos. Debido a su conveniencia, se lo utiliza para llevar a cabo el Certamen Escolar todos los años y el Torneo Jujeño de Programación. También es utilizado para desarrollar de forma virtual todos los años la OII (Olimpíada Iberoamericana de Informática).

¿Qué es omegaUp?

omegaUp es un juez online muy utilizado en México que permite fácilmente crear competencias y subir problemas nuevos con distintos formatos. Debido a su conveniencia, se lo utiliza para llevar a cabo el Certamen Escolar todos los años y el Torneo Jujeño de Programación. También es utilizado para desarrollar de forma virtual todos los años la OII (Olimpíada Iberoamericana de Informática).

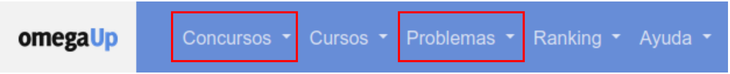
Por este motivo, vamos a familiarizarnos con el funcionamiento de los jueces online tomando como caso a omegaUp, para que se puedan manejar por la página en caso que deseen participar en las competencias que se preparan desde la escuela.

Login / crear una cuenta

¡Veámoslo en vivo!

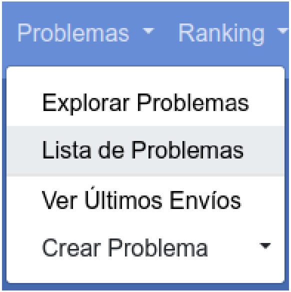
Página principal

Página principal

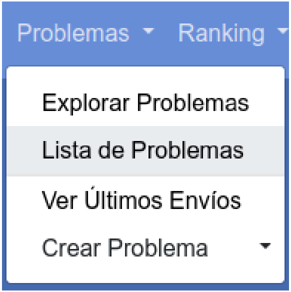


En la barra superior de la página, sólo nos van a interesar las secciones de "Problemas" y "Concursos".

Explorando problemas



Explorando problemas



Al hacer click en la pestaña de Problemas, podemos ver la lista de todos los problemas disponibles en el juez entrando a "Lista de Problemas".

Problemas

Buscador de Problemas

Buscar por alias, título o id de problema | Filtrar por idioma | Solo problemas de calidad | Buscar

ID	Título Nivel Etiqueta del autor Etiqueta de coders	Calidad	Dificultad	Ratio	Mi puntaje	
22390	¿PERDONAR o MATAR? Lenguaje Nivel Intermedio: Estructuras de datos y Algoritmos Conectividad en grafos Caminos más cortos	—	—	0.00% (0/1)	0.00	100.00
22376	Juego-Piedras Lenguaje Nivel Intermedio: Estructuras de datos y Algoritmos Árboles de segmentos	—	—	0.00% (0/14)	0.00	100.00

Al ingresar al listado, nos encontraremos en una pantalla como esta. Los problemas al principio son los subidos más recientemente.

Al ingresar al listado, nos encontraremos en una pantalla como esta. Los problemas al principio son los subidos más recientemente.

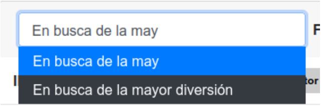
A la izquierda de cada problema aparece su **id**. Es muy común en los jueces online que cada problema tenga su id único.

Al ingresar al listado, nos encontraremos en una pantalla como esta. Los problemas al principio son los subidos más recientemente.
A la izquierda de cada problema aparece su **id**. Es muy común en los jueces online que cada problema tenga su id único.
En el caso de omegaUp, para buscar algún problema particular podemos usar el **buscador** arriba ingresando su nombre (título) o su id, y haciendo click en “Buscar”.

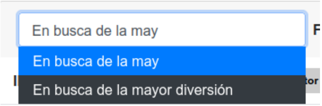
Si recordamos, ya habíamos visto un problema de omegaUp en la sección de complejidad...

Si recordamos, ya habíamos visto un problema de omegaUp en la sección de complejidad... probemos buscarlo por nombre en el buscador a ver si aparece.

Si recordamos, ya habíamos visto un problema de omegaUp en la sección de complejidad... probemos buscarlo por nombre en el buscador a ver si aparece.

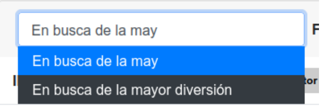


Si recordamos, ya habíamos visto un problema de omegaUp en la sección de complejidad... probemos buscarlo por nombre en el buscador a ver si aparece.

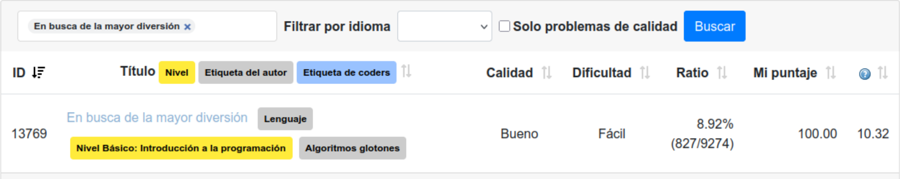


Como es de esperarse, aparece listado incluso mientras escribimos su título.

Si recordamos, ya habíamos visto un problema de omegaUp en la sección de complejidad... probemos buscarlo por nombre en el buscador a ver si aparece.



Como es de esperarse, aparece listado incluso mientras escribimos su título.



En busca de la mayor diversión × | Filtrar por idioma | Solo problemas de calidad | Buscar

ID	Título Nivel Etiqueta del autor Etiqueta de coders	Calidad	Dificultad	Ratio	Mi puntaje	
13769	En busca de la mayor diversión Lenguaje Nivel Básico: Introducción a la programación Algoritmos glotones	Bueno	Fácil	8.92% (827/9274)	100.00	10.32

En busca de la mayor diversión × | Filtrar por idioma | Solo problemas de calidad | Buscar

ID	Título Nivel Etiqueta del autor Etiqueta de coders	Calidad	Dificultad	Ratio	Mi puntaje	
13769	En busca de la mayor diversión Lenguaje Nivel Básico: Introducción a la programación Algoritmos glotones	Bueno	Fácil	8.92% (827/9274)	100.00	10.32

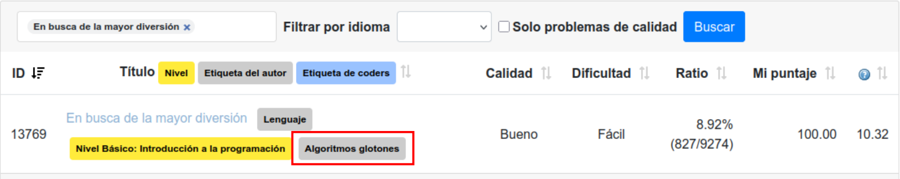
Podemos identificar algunos detalles más que aparecen sobre el problema:

En busca de la mayor diversión × Filtrar por idioma ☐ Solo problemas de calidad Buscar

ID	Título Nivel Etiqueta del autor Etiqueta de coders	Calidad	Dificultad	Ratio	Mi puntaje	?
13769	En busca de la mayor diversión Lenguaje Nivel Básico: Introducción a la programación Algoritmos glotones	Bueno	Fácil	8.92% (827/9274)	100.00	10.32

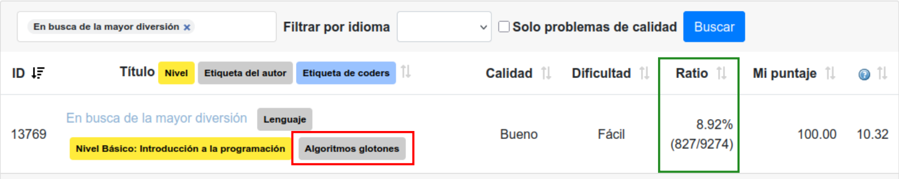
Podemos identificar algunos detalles más que aparecen sobre el problema:

- Técnicas / algoritmos relacionados con una posible solución.



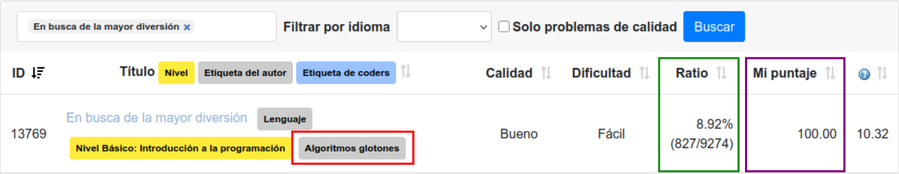
Podemos identificar algunos detalles más que aparecen sobre el problema:

- **Técnicas / algoritmos relacionados con una posible solución.** *Esto puede spoilearnos la solución!* Muchos jueces permiten ocultar las etiquetas. En omegaUp ir a Mi perfil → Preferencias (o en este link).



Podemos identificar algunos detalles más que aparecen sobre el problema:

- Técnicas / algoritmos relacionados con una posible solución. *Esto puede spoilearnos la solución!* Muchos jueces permiten ocultar las etiquetas. En omegaUp ir a Mi perfil → Preferencias (o en este link).
- El ratio: el porcentaje de envíos correctos respecto al total intentados por el resto de los usuarios de la página. Debajo nos indica `(aceptados / total)`.



Podemos identificar algunos detalles más que aparecen sobre el problema:

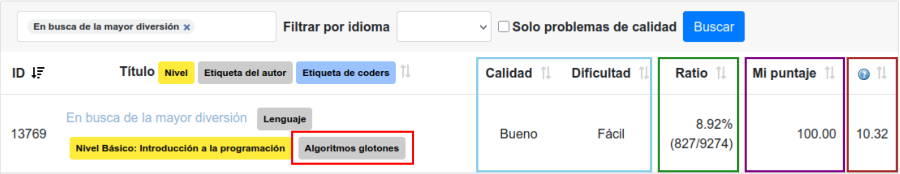
- **Técnicas / algoritmos relacionados con una posible solución**. *Esto puede spoilearnos la solución!* Muchos jueces permiten ocultar las etiquetas. En omegaUp ir a Mi perfil → Preferencias (o en este link).
- El **ratio**: el porcentaje de envíos correctos respecto al total intentados por el resto de los usuarios de la página. Debajo nos indica `(aceptados / total)`.
- **Mi puntaje**, el puntaje máximo obtenido en el problema. A mí me marca `100.00` porque ya lo resolví...

En busca de la mayor diversión × | Filtrar por idioma | Solo problemas de calidad | Buscar

ID	Título Nivel Etiqueta del autor Etiqueta de coders	Calidad	Dificultad	Ratio	Mi puntaje	
13769	En busca de la mayor diversión Lenguaje Nivel Básico: Introducción a la programación Algoritmos glotones	Bueno	Fácil	8.92% (827/9274)	100.00	10.32

Podemos identificar algunos detalles más que aparecen sobre el problema:

- **Técnicas / algoritmos relacionados con una posible solución**. *Esto puede spoilearnos la solución!* Muchos jueces permiten ocultar las etiquetas. En omegaUp ir a Mi perfil → Preferencias (o en este link).
- El **ratio**: el porcentaje de envíos correctos respecto al total intentados por el resto de los usuarios de la página. Debajo nos indica `(aceptados / total)`.
- **Mi puntaje**, el puntaje máximo obtenido en el problema. A mí me marca `100.00` porque ya lo resolví...
- El **nivel de calidad y dificultad** del problema. Poco común en otros jueces. Dado por votos de otros usuarios.



Podemos identificar algunos detalles más que aparecen sobre el problema:

- Técnicas / algoritmos relacionados con una posible solución. *Esto puede spoilearnos la solución!* Muchos jueces permiten ocultar las etiquetas. En omegaUp ir a Mi perfil → Preferencias (o en este link).
- El ratio: el porcentaje de envíos correctos respecto al total intentados por el resto de los usuarios de la página. Debajo nos indica `(aceptados / total)`.
- Mi puntaje, el puntaje máximo obtenido en el problema. A mí me marca `100.00` porque ya lo resolví...
- El nivel de calidad y dificultad del problema. Poco común en otros jueces. Dado por votos de otros usuarios.
- Un número de ranking de problema de omegaUp. No nos interesa mucho a nosotros.

TODO: Mostrar pantalla concursos, ir a pasados, buscar certamen, entrar al contest y explorar. Ver capaz algún concurso en vivo y mandar? Mostrar link modo práctica si hubiéramos entrado por lista de problemas. Explicar scoreboard.

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez.

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema

13769. En busca de la mayor diversión

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema

   13769. En busca de la mayor diversión

2. **Límite de tiempo** (por caso):

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema

   13769. En busca de la mayor diversión

2. **Límite de tiempo** (por caso): Cuando el programa es evaluado en el servidor, se lo suele ejecutar con ciertas entradas predefinidas llamados **casos de prueba** (salvo excepciones, ver "problemas interactivos") y se le da un cierto tiempo límite que puede tardar el programa en terminar de ejecutarse para cada entrada dada.

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema

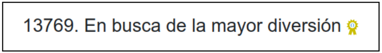
   13769. En busca de la mayor diversión

2. **Límite de tiempo** (por caso): Cuando el programa es evaluado en el servidor, se lo suele ejecutar con ciertas entradas predefinidas llamados **casos de prueba** (salvo excepciones, ver "problemas interactivos") y se le da un cierto tiempo límite que puede tardar el programa en terminar de ejecutarse para cada entrada dada.
   A este límite lo llamamos el *límite de tiempo*, o en inglés, **time limit** (dado que muchos jueces online están en inglés), abreviado a veces *TL*.

Estructura de un problema

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez. Entremos al enunciado del problema de antes y veamos los aspectos más importantes:

1. **Título** / nombre del problema



2. **Límite de tiempo** (por caso): Cuando el programa es evaluado en el servidor, se lo suele ejecutar con ciertas entradas predefinidas llamados **casos de prueba** (salvo excepciones, ver "problemas interactivos") y se le da un cierto tiempo límite que puede tardar el programa en terminar de ejecutarse para cada entrada dada.
   A este límite lo llamamos el *límite de tiempo*, o en inglés, **time limit** (dado que muchos jueces online están en inglés), abreviado a veces *TL*.
   Es concretamente por este límite que es tan importante el análisis de complejidad de programas que vimos antes.

2. **Límite de tiempo** (por caso): (cont.) En el problema:

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso):

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

   En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso): De forma análoga al time limit, existe el **memory limit** (a veces abreviado *ML*) o *límite de memoria*, que es la cantidad de memoria que el programa puede reservar durante la ejecución de cada caso de prueba.

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

   En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso): De forma análoga al time limit, existe el **memory limit** (a veces abreviado *ML*) o *límite de memoria*, que es la cantidad de memoria que el programa puede reservar durante la ejecución de cada caso de prueba. En el problema:

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

   En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso): De forma análoga al time limit, existe el **memory limit** (a veces abreviado *ML*) o *límite de memoria*, que es la cantidad de memoria que el programa puede reservar durante la ejecución de cada caso de prueba. En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

   En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso): De forma análoga al time limit, existe el **memory limit** (a veces abreviado *ML*) o *límite de memoria*, que es la cantidad de memoria que el programa puede reservar durante la ejecución de cada caso de prueba. En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

   Es decir, en este problema, al correr cada caso nuestra solución puede reservar a lo sumo 4.8828125 MiB.

2. **Límite de tiempo** (por caso): (cont.) En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

En este problema, cada caso tiene un tiempo límite de 1s para ejecutarse. De lo contrario, la solución se considerará **incorrecta**. En breve veremos cómo el juez no comunica esto.

3. **Límite de memoria** (por caso): De forma análoga al time limit, existe el **memory limit** (a veces abreviado *ML*) o *límite de memoria*, que es la cantidad de memoria que el programa puede reservar durante la ejecución de cada caso de prueba. En el problema:

Puntos	10.32	**Límite de memoria**	4.8828125 MiB
Límite de tiempo (caso)	1s	**Límite de tiempo (total)**	1m0s
Tamaño límite de entrada (bytes)	10 KiB		

Es decir, en este problema, al correr cada caso nuestra solución puede reservar a lo sumo 4.8828125 MiB. Número raro...

4. **Descripción del problema**:

4. **Descripción del problema:**

Descripción

Nicolás es un niño muy caprichoso y sabe que su mamá hará lo posible por mantenerlo feliz. Nicolás suele aprovecharse de este hecho (por favor, no seas como Nicolás).

Cada vez que Nicolás y su mamá van a hacer las compras, el niño exige que se le compren N juguetes. Normalmente termina ocurriendo, pero esta vez, la madre se puso firme y le dijo a Nicolás que no le comprará N juguetes, sino solamente $N - 1$. Es decir, si siempre le compraba 5 juguetes, esta vez le compraría solo 4.

Nicolás está en una situación difícil y necesita conseguir la mayor diversión posible. Cada juguete tiene un nivel de diversión y la diversión final es la suma de los niveles de diversión de los juguetes adquiridos. Tu trabajo es conseguir la mayor diversión posible.

4. **Descripción del problema:**

Descripción

Nicolás es un niño muy caprichoso y sabe que su mamá hará lo posible por mantenerlo feliz. Nicolás suele aprovecharse de este hecho (por favor, no seas como Nicolás).

Cada vez que Nicolás y su mamá van a hacer las compras, el niño exige que se le compren N juguetes. Normalmente termina ocurriendo, pero esta vez, la madre se puso firme y le dijo a Nicolás que no le comprará N juguetes, sino solamente $N - 1$. Es decir, si siempre le compraba 5 juguetes, esta vez le compraría solo 4.

Nicolás está en una situación difícil y necesita conseguir la mayor diversión posible. Cada juguete tiene un nivel de diversión y la diversión final es la suma de los niveles de diversión de los juguetes adquiridos. Tu trabajo es conseguir la mayor diversión posible.

5. **Entrada:**

4. **Descripción del problema**:

Descripción

Nicolás es un niño muy caprichoso y sabe que su mamá hará lo posible por mantenerlo feliz. Nicolás suele aprovecharse de este hecho (por favor, no seas como Nicolás).

Cada vez que Nicolás y su mamá van a hacer las compras, el niño exige que se le compren N juguetes. Normalmente termina ocurriendo, pero esta vez, la madre se puso firme y le dijo a Nicolás que no le comprará N juguetes, sino solamente $N-1$. Es decir, si siempre le compraba 5 juguetes, esta vez le compraría solo 4.

Nicolás está en una situación difícil y necesita conseguir la mayor diversión posible. Cada juguete tiene un nivel de diversión y la diversión final es la suma de los niveles de diversión de los juguetes adquiridos. Tu trabajo es conseguir la mayor diversión posible.

5. **Entrada**: Suele incluir detalles específicos sobre cómo se provee la información del problema al programa que implementemos.

4. **Descripción del problema**:

    ### Descripción

    Nicolás es un niño muy caprichoso y sabe que su mamá hará lo posible por mantenerlo feliz. Nicolás suele aprovecharse de este hecho (por favor, no seas como Nicolás).

    Cada vez que Nicolás y su mamá van a hacer las compras, el niño exige que se le compren N juguetes. Normalmente termina ocurriendo, pero esta vez, la madre se puso firme y le dijo a Nicolás que no le comprará N juguetes, sino solamente $N-1$. Es decir, si siempre le compraba 5 juguetes, esta vez le compraría solo 4.

    Nicolás está en una situación difícil y necesita conseguir la mayor diversión posible. Cada juguete tiene un nivel de diversión y la diversión final es la suma de los niveles de diversión de los juguetes adquiridos. Tu trabajo es conseguir la mayor diversión posible.

5. **Entrada**: Suele incluir detalles específicos sobre cómo se provee la información del problema al programa que implementemos. Dependiendo del juez / origen, también puede contener las **cotas** del problema, que en este caso se encuentran por separado más abajo.

4. **Descripción del problema**:

Descripción

Nicolás es un niño muy caprichoso y sabe que su mamá hará lo posible por mantenerlo feliz. Nicolás suele aprovecharse de este hecho (por favor, no seas como Nicolás).

Cada vez que Nicolás y su mamá van a hacer las compras, el niño exige que se le compren N juguetes. Normalmente termina ocurriendo, pero esta vez, la madre se puso firme y le dijo a Nicolás que no le comprará N juguetes, sino solamente $N-1$. Es decir, si siempre le compraba 5 juguetes, esta vez le compraría solo 4.

Nicolás está en una situación difícil y necesita conseguir la mayor diversión posible. Cada juguete tiene un nivel de diversión y la diversión final es la suma de los niveles de diversión de los juguetes adquiridos. Tu trabajo es conseguir la mayor diversión posible.

5. **Entrada**: Suele incluir detalles específicos sobre cómo se provee la información del problema al programa que implementemos. Dependiendo del juez / origen, también puede contener las **cotas** del problema, que en este caso se encuentran por separado más abajo.

Entrada

- Una línea con la cantidad N de juguetes que eligió inicialmente Nicolás.
- N líneas, cada una con el nivel de diversión de un juguete.

6. **Salida**:

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como "qué solución imprimir" si es que hay varias o qué imprimir si una solución "no existe" (según el problema).

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como "qué solución imprimir" si es que hay varias o qué imprimir si una solución "no existe" (según el problema).

Salida

Una línea con la mayor diversión posible habiendo dejado exactamente un juguete sin comprar (puede ser cualquiera, siempre y cuando la diversión obtenida sea la mayor posible).

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como "qué solución imprimir" si es que hay varias o qué imprimir si una solución "no existe" (según el problema).

   Salida

   Una línea con la mayor diversión posible habiendo dejado exactamente un juguete sin comprar (puede ser cualquiera, siempre y cuando la diversión obtenida sea la mayor posible).

7. **Ejemplo(s)**:

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como “qué solución imprimir” si es que hay varias o qué imprimir si una solución “no existe” (según el problema).

   > **Salida**
   >
   > Una línea con la mayor diversión posible habiendo dejado exactamente un juguete sin comprar (puede ser cualquiera, siempre y cuando la diversión obtenida sea la mayor posible).

7. **Ejemplo(s)**: Casi siempre los problemas contienen uno o más ejemplos (**samples**, en inglés) de entrada y salida válidos para una solución del problema, con el fin de comprender correctamente el formato de ambos o incluso aclarar el enunciado.

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como "qué solución imprimir" si es que hay varias o qué imprimir si una solución "no existe" (según el problema).

Salida

Una línea con la mayor diversión posible habiendo dejado exactamente un juguete sin comprar (puede ser cualquiera, siempre y cuando la diversión obtenida sea la mayor posible).

7. **Ejemplo(s)**: Casi siempre los problemas contienen uno o más ejemplos (**samples**, en inglés) de entrada y salida válidos para una solución del problema, con el fin de comprender correctamente el formato de ambos o incluso aclarar el enunciado.

Ejemplo

Entrada	Salida	Descripción
5 8 5 3 6 8	27	Nicolás puede tomar los juguetes 1, 2, 4 y 5.

6. **Salida**: De forma análoga a la entrada, especifica detalles sobre cómo escribir el resultado de resolver el problema para la entrada dada. En algunos casos presenta información más importante como "qué solución imprimir" si es que hay varias o qué imprimir si una solución "no existe" (según el problema).

   Salida

   Una línea con la mayor diversión posible habiendo dejado exactamente un juguete sin comprar (puede ser cualquiera, siempre y cuando la diversión obtenida sea la mayor posible).

7. **Ejemplo(s)**: Casi siempre los problemas contienen uno o más ejemplos (**samples**, en inglés) de entrada y salida válidos para una solución del problema, con el fin de comprender correctamente el formato de ambos o incluso aclarar el enunciado.

   Ejemplo

Entrada	Salida	Descripción
5 8 5 3 6 8	27	Nicolás puede tomar los juguetes 1, 2, 4 y 5.

En omegaUp también suelen incluir una *Descripción* al lado de los ejemplos, explicando los resultados presentados. Otros jueces suelen incluir esto en una sección separada llamada *Notas*.

6. **Cotas**:

6. **Cotas**:

Competencias (concursos)

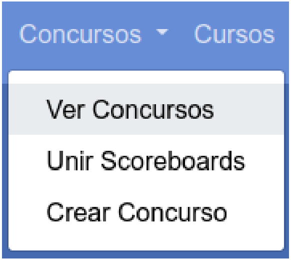
Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente.

Competencias (concursos)

Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.

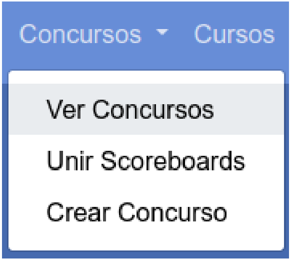
Competencias (concursos)

Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.



Competencias (concursos)

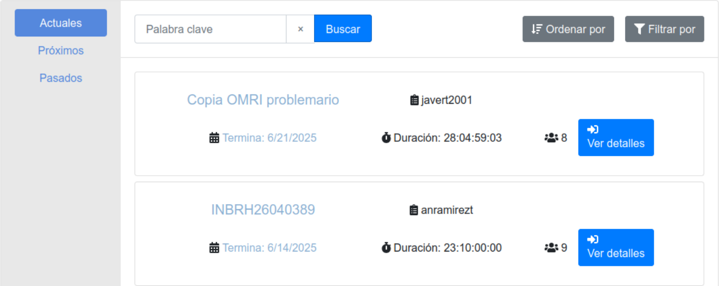
Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.



Al hacer click en la pestaña Concursos desde la página principal, podemos ver la lista de todos los concursos disponibles entrando a "Ver Concursos".

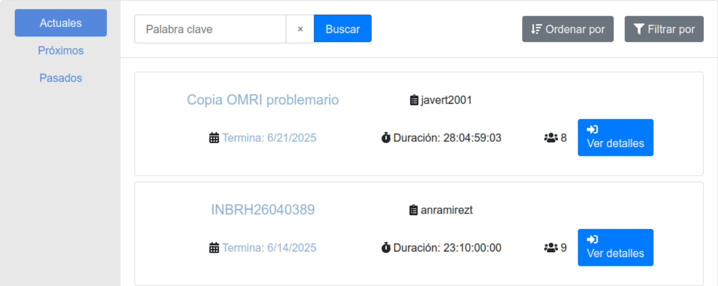

Concursos
Actuales
Próximos
Pasados
Palabra clave
×
Buscar
Ordenar por
Filtrar por
Copia OMRI problemario
javert2001
Termina: 6/21/2025
Duración: 28:04:59:03
8
Ver detalles
INBRH26040389
anramirezt
Termina: 6/14/2025
Duración: 23:10:00:00
9
Ver detalles

Concursos



Al ingresar, nos encontraremos en una pantalla como esta. Aquí se encuentran las competencias en curso, ordenadas por tiempo de finalización de “más tarde a más temprano”. Hay que considerar que una competencia se puede armar para que dure días o incluso meses.

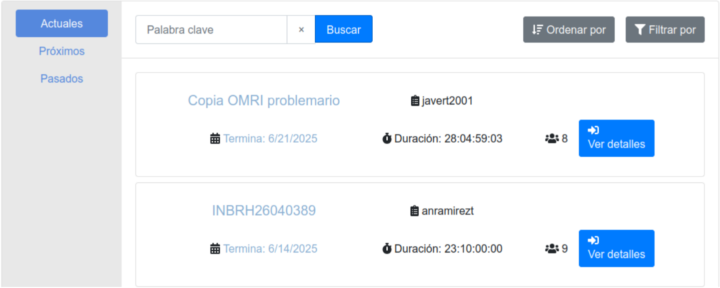
Concursos



Al ingresar, nos encontraremos en una pantalla como esta. Aquí se encuentran las competencias en curso, ordenadas por tiempo de finalización de "más tarde a más temprano". Hay que considerar que una competencia se puede armar para que dure días o incluso meses.

Como podemos ver, hay muchas competencias no relacionadas creadas por usuarios del sitio, en muchos casos para sus propios propósitos.

Concursos



Al ingresar, nos encontraremos en una pantalla como esta. Aquí se encuentran las competencias en curso, ordenadas por tiempo de finalización de "más tarde a más temprano". Hay que considerar que una competencia se puede armar para que dure días o incluso meses.
Como podemos ver, hay muchas competencias no relacionadas creadas por usuarios del sitio, en muchos casos para sus propios propósitos.
Dependiendo la competencia, también puede requerirse aprobación del autor para participar (por ejemplo, para la OII).

TODO: Buscar ToJuPro? O algún otro? Mostrar en vivo sino?