Complejidad de Algoritmos y Jueces Online

Ariel Leonardo Fideleff

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

31 de mayo del 2025

- Introducción
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

- Introducción
- 2 Complejidad computaciona
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

¡Hola!

- Introducción
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

- Introducción
- Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

Veamos el siguiente código en C++:

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
}</pre>
```

Veamos el siguiente código en C++:

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
}</pre>
```

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

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
}</pre>
```

¿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.

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
}</pre>
```

¿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.

Ordenar números es fácil, no?

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

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
    int asz = mid-1, bsz = r-mid;
    vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
        a[i] = arr[1+i];
    for (int i = 0; i < bsz; i++)
        b[i] = arr[mid+i];
    int aidx = 0, bidx = 0, idx = 1;
    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++;
    7
}
```

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

    int mid = (r+1)/2;
    split(arr,1,mid);
    split(arr,mid,r);
    merge(arr,1,mid,r);
}

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

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
    int asz = mid-1, bsz = r-mid;
   vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
       a[i] = arr[1+i];
    for (int i = 0; i < bsz; i++)
       b[i] = arr[mid+i];
                                                             void split(vector<int> &arr, int 1, int r) {
                                                                 if (r-1 <= 1)
    int aidx = 0, bidx = 0, idx = 1;
                                                                      return:
    while (aidx < asz %% bidx < bsz)
       if (a[aidx] <= b[bidx]) {
           arr[idx] = a[aidx];
                                                                  int mid = (r+1)/2;
           idx++, aidx++;
                                                                  split(arr.l.mid):
       else {
                                                                  split(arr,mid,r);
           arr[idx] = b[bidx];
                                                                 merge(arr,1,mid,r);
           idx++, bidx++:
    while (aidx < asz) {
                                                             void sort(vector<int> &arr, int n) {
       arr[idx] = a[aidx];
       idx++, aidx++;
                                                                  split(arr.0.n):
    while (bidx < bsz) {
       arr[idx] = b[bidx];
       idx++, bidx++:
   7
}
```

Mucho código...

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
    int asz = mid-1, bsz = r-mid;
   vector<int> a(asz), b(bsz);
    for (int i = 0; i < asz; i++)
       a[i] = arr[1+i];
    for (int i = 0; i < bsz; i++)
       b[i] = arr[mid+i];
                                                             void split(vector<int> &arr, int 1, int r) {
                                                                 if (r-1 <= 1)
    int aidx = 0, bidx = 0, idx = 1;
                                                                      return:
    while (aidx < asz %% bidx < bsz)
       if (a[aidx] <= b[bidx]) {
           arr[idx] = a[aidx];
                                                                  int mid = (r+1)/2;
           idx++, aidx++;
                                                                  split(arr.l.mid):
       else {
                                                                  split(arr,mid,r);
           arr[idx] = b[bidx];
                                                                 merge(arr,1,mid,r);
           idx++, bidx++:
    while (aidx < asz) {
                                                             void sort(vector<int> &arr, int n) {
       arr[idx] = a[aidx];
       idx++, aidx++;
                                                                  split(arr.0.n):
    while (bidx < bsz) {
       arr[idx] = b[bidx];
       idx++, bidx++:
   }
}
```

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

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
   int asz = mid-1, bsz = r-mid;
   vector<int> a(asz), b(bsz);
   for (int i = 0; i < asz; i++)
       a[i] = arr[1+i];
   for (int i = 0; i < bsz; i++)
       b[i] = arr[mid+i];
                                                             void split(vector<int> &arr, int 1, int r) {
                                                                 if (r-1 <= 1)
   int aidx = 0, bidx = 0, idx = 1;
                                                                      return:
   while (aidx < asz %% bidx < bsz)
       if (a[aidx] <= b[bidx]) {
           arr[idx] = a[aidx];
                                                                 int mid = (r+1)/2;
           idx++, aidx++;
                                                                 split(arr.l.mid):
       else {
                                                                 split(arr,mid,r);
           arr[idx] = b[bidx];
                                                                 merge(arr,1,mid,r);
           idx++, bidx++;
   while (aidx < asz) {
                                                             void sort(vector<int> &arr, int n) {
       arr[idx] = a[aidx];
       idx++, aidx++;
                                                                 split(arr.0.n):
   while (bidx < bsz) {
       arr[idx] = b[bidx];
       idx++, bidx++:
   }
}
```

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

Este algoritmo también es conocido, se llama merge sort.

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

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.

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)

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

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)

(a) Merge Sort



(b) Selection Sort

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



(b) Selection Sort

Parece que una de las dos se toma su tiempo.

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



(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.

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



(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?

- Introducción
- Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

\mathcal{O} qué!?

En Programación Competitiva nos gusta que las cosas sean rápidas.

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).

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

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]".

mirarlo un poco.

¿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.

${\it i}$ Cómo L ${\it O}$ 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 $\mathcal{O}(n)$.

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.

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.

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.

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.

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 $\mathcal{O}(n)$.

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 $\mathcal{O}(n)$.

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;
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;</pre>
```

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 $\mathcal{O}(n)$.

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 $\mathcal{O}(n)$.

Definición (notación \mathcal{O} , informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un nsuficientemente 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;
for (int i = 0; i < n; i++) \rightarrow n operaciones (leemos n números)
    cin >> arr[i]:
int sum = 0:
for (int i = 0; i < n; i++) \rightarrow n operaciones (sumamos n números)
    sum += arr[i];
cout << sum;
Hicimos entonces 2n operaciones. O sea, c * n con c = 2
```

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:

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

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; \rightarrow 1 operación (leemos 1 número)

for (int i = 0; i < n; i++) \rightarrow n operaciones (leemos n números)

cin >> arr[i];

int sum = 0;

for (int i = 0; i < n; i++) \rightarrow n operaciones (sumamos n números)

sum += arr[i];

cout << sum; \rightarrow 1 operación (imprimimos 1 número)

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

Definición (notación \mathcal{O} , informal)

Sea un programa que recibe una cantidad de datos n. Si a partir de un nsuficientemente 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; \rightarrow 1 operación (leemos 1 número) for (int i = 0; i < n; i++) \rightarrow n operaciones (leemos n números)
     cin >> arr[i]:
int sum = 0:
for (int i = 0; i < n; i++) \rightarrow n operaciones (sumamos n números)
     sum += arr[i];
                                     \rightarrow 1 operación (imprimimos 1 número)
cout << sum;
Hicimos entonces 2n + 2 operaciones. 2n + 2 < 2n + n = 3n para n > 2.
```

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 $\mathcal{O}(n)$.

Hicimos entonces 2n + 2 operaciones. $2n + 2 \le 2n + n = 3n$ para $n \ge 2$.

Por lo tanto, este programa corre en $\mathcal{O}(n)$, pues ejecuta a lo sumo 3n operaciones a partir de un n "suficientemente grande".

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

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.

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 $0 \le f(n) \le c.g(n), \forall n \ge n_0$.

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 $0 \le f(n) \le c.g(n), \ \forall n \ge n_0$.

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.

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 $0 \le f(n) \le c.g(n), \ \forall n \ge n_0$.

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.

- Introducción
- Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

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.

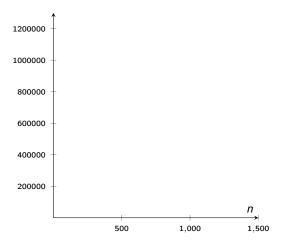
Si bien técnicamente por definición podríamos decir que un programa "corre en $\mathcal{O}(2n)$ " en vez de decir que corre en $\mathcal{O}(n)$, no tiene mucho sentido hacerlo porque lo que más afecta a la rapidez de nuestro programa para una entrada suficientemente grande es la cantidad de operaciones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

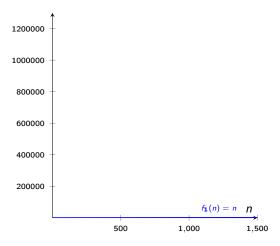
Una constante no deja de ser constante

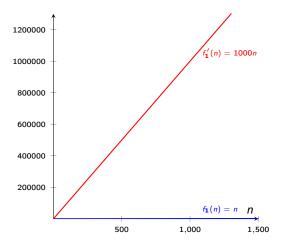
Como vimos en el ejemplo de la sección anterior, las constantes no afectan a la cota \mathcal{O} de un programa.

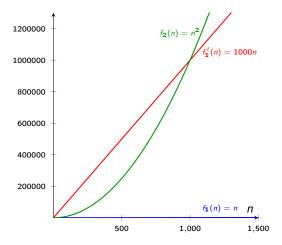
Si bien técnicamente por definición podríamos decir que un programa "corre en $\mathcal{O}(2n)$ " en vez de decir que corre en $\mathcal{O}(n)$, no tiene mucho sentido hacerlo porque lo que más afecta a la rapidez de nuestro programa para una entrada suficientemente grande es la cantidad de operaciones 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:









Así, en la práctica observamos que:

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

- 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.

- 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.

- 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:</pre>
```

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.

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.

- Introducción
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

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

- 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.

- 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 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)).

- 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 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.

- 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 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.

- 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 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.

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.

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.

Problema (Watermelon)

Problema (Watermelon, CodeForces 4A)

Problema (Watermelon, CodeForces 4A)

(Juez Online, más adelante)

Problema (Watermelon, CodeForces 4A)

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

Problema (Watermelon, CodeForces 4A)

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

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

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.

Problema (Watermelon, CodeForces 4A)

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

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

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

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.

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.

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.

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso par. Dado w, jes 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;
                                 \leftarrow 1 operación (leer un número)
if ((w\%2) == 0)
                                 \leftarrow 2 operaciones (sacar resto %2 y comparar con 0)
                                 \leftarrow 1 operación (imprimir la respuesta)
     cout << "YES":
else
     cout << "NO":
                                 \leftarrow 1 operación (imprimir la respuesta)
```

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

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso par. Dado w, jes 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;
                                 \leftarrow 1 operación (leer un número)
if ((w\%2) == 0)
                                 \leftarrow 2 operaciones (sacar resto %2 y comparar con 0)
                                 \leftarrow 1 operación (imprimir la respuesta)
     cout << "YES":
else
     cout << "NO":
                                 \leftarrow 1 operación (imprimir la respuesta)
```

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

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso par. Dado w, jes 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;
                                 \leftarrow 1 operación (leer un número)
if ((w\%2) == 0)
                                 \leftarrow 2 operaciones (sacar resto %2 y comparar con 0)
                                 \leftarrow 1 operación (imprimir la respuesta)
     cout << "YES":
else
     cout << "NO":
                                 \leftarrow 1 operación (imprimir la respuesta)
```

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

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.

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 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 $a \neq b$ differen en una *constante*.

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

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

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

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

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

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

$$\frac{\log_a n}{\log_a b} = \log_b n \Rightarrow \log_a n = \underbrace{\log_a b}_{\text{constante}} \cdot \log_b n$$

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

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 $n \le \log n$ para $n \ge 0$, el programa correrá en al menos $\mathcal{O}(n)$.

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...

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 $n \le \log n$ para $n \ge 0$, 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......

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 $n \le \log n$ para $n \ge 0$, 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!

Problema (búsqueda binaria)

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.

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.

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-1)/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 $(-1, \lfloor \frac{n-1}{2} \rfloor]$, descartando los últimos $\left|\frac{n}{2}\right|$ elementos.

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-1)/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 $(-1, \lfloor \frac{n-1}{2} \rfloor]$, descartando los últimos $\lfloor \frac{n}{2} \rfloor$ 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 $(\lfloor \frac{n-1}{2} \rfloor, n)$, descartando los primeros $\lceil \frac{n}{2} \rceil$ elementos.

```
int x;
vector<int> arr(n); // Suponemos que está ordenado
// ...
int lo = -1, hi = n;
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].
// Si no existe, termina con hi = n.
```

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

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).

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.

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

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 O: sumar los números de una lista.

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 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.

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

Problema (En busca de la mayor diversión)

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

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

(Juez Online)

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.

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.

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 Njuguetes, 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.

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 Njuguetes, 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++) // Leemos los números
    cin >> arr[i];
```

```
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leemos 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]);</pre>
```

```
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leemos 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</pre>
```

```
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leemos 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</pre>
```

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

```
int n; cin >> n;
vector<int> arr(n);
for (int i = 0; i < n; i++) // Leemos 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</pre>
```

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...

```
int n; cin >> n;
vector<int> arr(n):
for (int i = 0; i < n; i++) // Leemos 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 **iuez**.

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.

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

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 $n \log n$.

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.

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 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):

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

• 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).

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>
 - Si vale, entonces el siguiente elemento de arr debe ser a[aidx].

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>
 - Si vale, entonces el siguiente elemento de arr debe ser a[aidx].
 - Sino, debe ser b[bidx].

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>
 - 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.

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>
 - 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.

Se repite el paso anterior hasta que alguno de los arreglos no tenga más elementos.

- 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).
- En cada paso se compara a[aidx] <= b[bidx].</p>
 - 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.

- Se repite el paso anterior hasta que alguno de los arreglos no tenga más elementos.
- Finalmente, se escriben los elementos restantes de la mitad que todavía tenga elementos (si los hay) en arr.

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

```
void merge(vector<int> &arr, int 1, 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];</pre>
```

```
void merge(vector<int> &arr, int 1, 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];</pre>
```

```
asz + bsz operaciones
```

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

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];</pre>
```

n operaciones

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

```
int asz = mid-1, bsz = r-mid;
vector<int> a(asz), b(bsz);
for (int i = 0; i < asz; i++)
    a[i] = arr[1+i];
for (int i = 0; i < bsz; i++)
    b[i] = arr[mid+i];</pre>
```



```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }</pre>
```

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
   if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
   }
   else {
        arr[idx] = b[bidx];
        idx++, bidx++;
   }</pre>
```

a lo sumo asz + bsz iteraciones

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
    }</pre>
```

a lo sumo n iteraciones

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) {
        arr[idx] = a[aidx];
        idx++, aidx++;
    }
    else {
        arr[idx] = b[bidx];
        idx++, bidx++;
}</pre>
```

 $\mathcal{O}(n)$

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

```
int asz = mid-1, bsz = r-mid;
vector<int> a(asz), b(bsz);
                                     for (int i = 0; i < bsz; i++)
for (int i = 0; i < asz; i++)
                                          b[i] = arr[mid+i]:
    a[i] = arr[l+i]:
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];</pre>
    else arr[idx++] = b[bidx++];
```

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

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

a lo sumo max{asz,bsz} iteraciones

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

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

a lo sumo **n** iteraciones

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

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

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

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

 $\mathcal{O}(n)$

```
void merge(vector<int> &arr, int 1, int mid, int r) {
```

Se copian las mitades a arreglos auxiliares a y b.

```
\mathcal{O}(n)
```

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

```
int aidx = 0, bidx = 0, idx = 1;
while (aidx < asz && bidx < bsz)
    if (a[aidx] <= b[bidx]) arr[idx++] = a[aidx++];
    else arr[idx++] = b[bidx++];</pre>
```

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

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

}

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

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

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.

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:

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 1, int r) {
    if (r-1 <= 1)
        return;
    int mid = (r+1)/2:
    split(arr,1,mid);
    split(arr,mid,r);
    merge(arr,1,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 1, int r) {
    if (r-1 <= 1)
                          ← caso trivial (un sólo elemento)
        return;
    int mid = (r+1)/2:
    split(arr,1,mid);
    split(arr,mid,r);
    merge(arr,1,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 1, int r) {
    if (r-1 <= 1)
                          ← caso trivial (un sólo elemento)
        return;
    int mid = (r+1)/2;
    split(arr,1,mid); } ordenar mitades recursivamente
    split(arr,mid,r);
    merge(arr,1,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 1, int r) {
    if (r-1 <= 1)
                          ← caso trivial (un sólo elemento)
        return;
    int mid = (r+1)/2;
    split(arr,1,mid); } ordenar mitades recursivamente
    split(arr,mid,r);
    merge(arr,1,mid,r); ← combinar mitades en un arreglo ordenado
```

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:

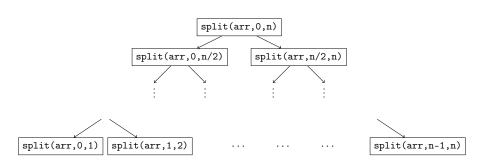
Notar que split ordena el intervalo [I, r) del arreglo, pasándolo cada vez por referencia para evitar costos adicionales.

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.

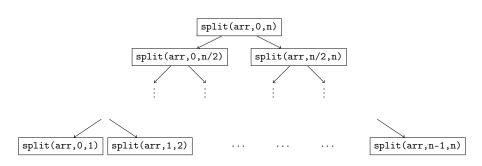
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial.

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 log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

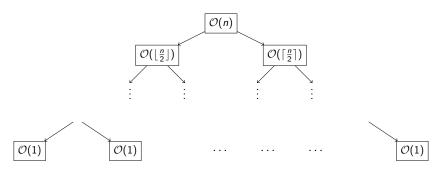
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:



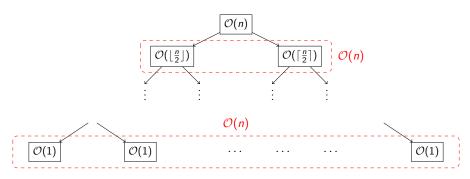
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:



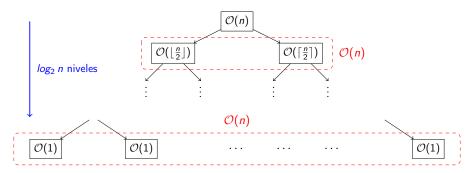
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:

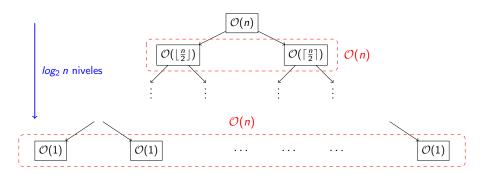


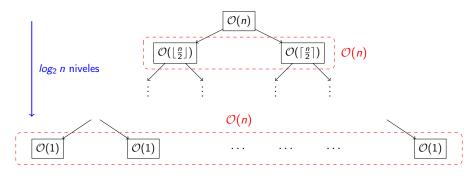
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:



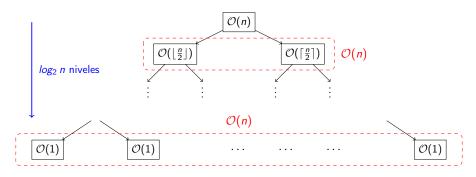
En efecto, en cada paso se llama split en ambas mitades del arreglo de igual tamaño, así queriendo ordenar la mitad de elementos de éste. Por definición de logaritmo, se harán alrededor de log₂ n llamadas en una cadena hasta llegar a un arreglo de un elemento, el caso trivial. Esto es:





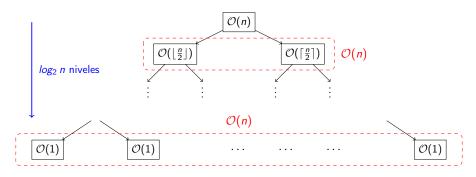


Así, como en cada nivel se hacen un total de operaciones del orden $\mathcal{O}(n)$, y la cadena de llamadas recursivas hecha por split alcanza a lo sumo una longitud de $log_2 n$, concluimos que split corre en $\mathcal{O}(n log n)$ para un arreglo de n elementos en su intervalo [0, n).



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).

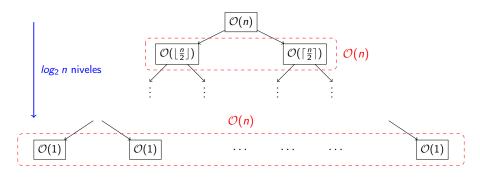
La última función que nos queda por analizar es sort, que simplemente llama a split de forma conveniente:



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

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.



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.

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

```
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]);
}</pre>
```

```
void sort(vector<int> &arr, int n) {
                                        \rightarrow n iteraciones
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++) \rightarrow n-i-1 iteraciones
             if (arr[i] < arr[i])</pre>
                  swap(arr[i],arr[j]);
```

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
                                               \rightarrow n iteraciones
    for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++) \rightarrow n-i-1 iteraciones
              if (arr[j] < arr[i])</pre>
                   swap(arr[i],arr[j]);
```

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

Recordemos la implementación de selection sort...

```
void sort(vector<int> &arr, int n) {
                                               \rightarrow n iteraciones
    for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++) \rightarrow n-i-1 iteraciones
              if (arr[i] < arr[i])</pre>
                   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.

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

$$f(n) = c(n-1) + c(n-2) + \dots + 2c + c$$
$$= \sum_{i=1}^{n-1} c(n-i)$$

$$f(n) = c(n-1) + c(n-2) + \dots + 2c + c$$
$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci$$

$$f(n) = c(n-1) + c(n-2) + \dots + 2c + c$$
$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2}$$

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2} = c \left(\frac{n^2}{2} - \frac{n}{2}\right)$$

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2} = c \left(\frac{n^2}{2} - \frac{n}{2}\right) \le c \frac{n^2}{2}$$

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2} = c \left(\frac{n^2}{2} - \frac{n}{2}\right) \le c \frac{n^2}{2} \le cn^2$$

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

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2} = c \left(\frac{n^2}{2} - \frac{n}{2}\right) \le c \frac{n^2}{2} \le cn^2$$

$$\therefore f(n) \in \mathcal{O}(n^2)$$

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

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

$$= \sum_{i=1}^{n-1} c(n-i) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i$$

$$= c \frac{n(n-1)}{2} = c \left(\frac{n^2}{2} - \frac{n}{2}\right) \le c \frac{n^2}{2} \le cn^2$$

$$\therefore f(n) \in \mathcal{O}(n^2)$$

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



Observación

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

Observación

- ullet Si bien obtuvimos una cota asintótica ${\cal O}$ 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

- ullet Si bien obtuvimos una cota asintótica ${\cal O}$ 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.

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:

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:

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

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 $f_i \in \mathcal{O}(g(x))$.

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.

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 $f_i \in \mathcal{O}(n) \not\Rightarrow f_i \in \mathcal{O}(1)$.

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.

- Introducción
- Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

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

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.

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 10⁸ operaciones corre en 1s.

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 10⁸ 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.

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 10⁸ 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:

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 10⁸ 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.

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.

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.

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:

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

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.

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.

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.

- 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:

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.

- 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: $\mathcal{O}(n \log n)$.

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.

- 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.

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.

- 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. Eiecuta aproximadamente $10^5 \log_2 10^5 \approx 1.7 \cdot 10^6$ operaciones.

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.

- 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.

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

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?

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.

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 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.

- 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.

- 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.

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

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.

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.

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.

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.

Sin mencionar que muchas otras optimizaciones menores hoy en día las realiza el compilador automáticamente.

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.

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.

- Introducción
- Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

Problema (Nearest Smaller Values)

Problema (Nearest Smaller Values, CSES 1645)

Problema (Nearest Smaller Values, CSES 1645)

(Juez Online)

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.

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. $1 < n < 2 \cdot 10^5$

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. $1 \le n \le 2 \cdot 10^5$

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. $1 \le n \le 2 \cdot 10^5$

¿Soluciones?

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

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. $1 < n < 2 \cdot 10^5$

¿Soluciones?

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

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. $1 < n < 2 \cdot 10^5$

¿Soluciones?

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. X, 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. \checkmark , pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos.

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. $1 < n < 2 \cdot 10^5$

- Recorrer el arreglo, y para cada elemento buscar por los anteriores hasta encontrar uno menor. $\mathcal{O}(n^2)$. X, 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. \checkmark , pero overkill.
- (avanzado) Compresión de la lista de nros + Segment Tree en rangos. $\mathcal{O}(n \log n)$.

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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. $1 < n < 2 \cdot 10^5$

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

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:

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

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

- Si arr[i-1] < arr[i], la respuesta es i-1.</p>
- 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.

- Si arr[i-1] < arr[i], la respuesta es i-1.</p>
- 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.
 - Si ese número es menor a arr[i], obtuvimos la respuesta.

- Si arr[i-1] < arr[i], la respuesta es i-1.</pre>
- 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.
 - Si ese número es menor a arr[i], obtuvimos la respuesta.
 - Sino, repetimos el razonamiento anterior para obtener el primero menor a izq del obtenido.

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:

- Si arr[i-1] < arr[i], la respuesta es i-1.</pre>
- 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.
 - Si ese número es menor a arr[i], obtuvimos la respuesta.
 - Sino, repetimos el razonamiento anterior para obtener el primero menor a iza 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].

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:

- ⑤ Si arr[i-1] < arr[i], la respuesta es i-1.</pre>
- 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.
 - Si ese número es menor a arr[i], obtuvimos la respuesta.
 - Sino, repetimos el razonamiento anterior para obtener el primero menor a izg 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].

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.

Analicemos la complejidad de la solución viendo su implementación:

Analicemos la complejidad de la solución viendo su implementación:

Analicemos la complejidad de la solución viendo su implementación:

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

Analicemos la complejidad de la solución viendo su implementación:

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



Analicemos la complejidad de la solución viendo su implementación:

• 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];

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 $1 \le arr[i] \le 10^9$)...

Analicemos la complejidad de la solución viendo su implementación:

● 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];

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 $1 \le arr[i] \le 10^9$)... arr[0] = -1:

Analicemos la complejidad de la solución viendo su implementación:

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];

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 $1 \le arr[i] \le 10^9$)...

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

Analicemos la complejidad de la solución viendo su implementación:

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

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 ≤ arr[i] ≤ 10⁹)... arr[0] = -1: ← constante
- 3 Para cada número en las posiciones [1, n]...

Analicemos la complejidad de la solución viendo su implementación:

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

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 ≤ arr[i] ≤ 10⁹)... arr[0] = -1; ← constante
- 3 Para cada número en las posiciones [1, n]... for (int i = 1; i <= n; i++) {

Analicemos la complejidad de la solución viendo su implementación:

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

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 ≤ arr[i] ≤ 10⁹)... arr[0] = -1; ← constante
- 3 Para cada número en las posiciones [1, n]... for (int i = 1; i <= n; i++) { $\leftarrow n$ iteraciones

Analicemos la complejidad de la solución viendo su implementación:

Leemos la entrada en posiciones 1 a n.

```
int n: cin >> n:
                                                      \mathcal{O}(n)
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

- 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 $1 \le arr[i] \le 10^9$)... arr[0] = -1: \leftarrow constante
- 3 Para cada número en las posiciones [1, n]... for (int i = 1; $i \le n$; i++) { $\leftarrow n$ iteraciones

...obtenemos su primero menor a izg como describimos en la idea...

Analicemos la complejidad de la solución viendo su implementación:

● 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];

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 ≤ arr[i] ≤ 10⁹)... arr[0] = -1; ← 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...

Analicemos la complejidad de la solución viendo su implementación:

● 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];

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 ≤ arr[i] ≤ 10⁹)...
 arr[0] = -1: ← constante
- 3 Para cada número en las posiciones [1, n]...

```
for (int i = 1; i \le n; i++) { \leftarrow n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

Analicemos la complejidad de la solución viendo su implementación:

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];

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 ≤ arr[i] ≤ 10⁹)...
 arr[0] = -1: ← 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 izg como describimos en la idea...

Analicemos la complejidad de la solución viendo su implementación:

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];

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 ≤ arr[i] ≤ 10⁹)... arr[0] = -1: ← 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 izg como describimos en la idea...

...y finalmente lo escribimos a la salida.

Analicemos la complejidad de la solución viendo su implementación:

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];

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 ≤ arr[i] ≤ 10⁹)...
 arr[0] = -1: ← constante
- $oldsymbol{\circ}$ Para cada número en las posiciones [1, n]...

```
for (int i = 1; i \le n; i++) { \leftarrow n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

...y finalmente lo escribimos a la salida.

```
}
```

Analicemos la complejidad de la solución viendo su implementación:

Leemos la entrada en posiciones 1 a n.

```
int n: cin >> n:
                                                      \mathcal{O}(n)
vector<int> arr(n+1), ans(n+1);
for(int i = 1; i <= n; i++) cin >> arr[i];
```

- 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 $1 \le arr[i] \le 10^9$)... 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 izg como describimos en la idea...

```
int i = i-1:
                                ← constante
while (arr[j] >= arr[i])
    j = ans[j];
ans[i] = j;
                               ← constante
```

...y finalmente lo escribimos a la salida. cout << j << ' ';

} Ariel Fideleff

Analicemos la complejidad de la solución viendo su implementación:

● 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];

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 ≤ arr[i] ≤ 10⁹)...
 arr[0] = -1: ← constante
- $oldsymbol{\circ}$ Para cada número en las posiciones [1, n]...

```
for (int i = 1; i \le n; i++) { \leftarrow n iteraciones
```

...obtenemos su primero menor a izq como describimos en la idea...

```
int j = i-1; \leftarrow constante while (arr[j] >= arr[i]) j = ans[j]; \rightarrow constante \mathcal{O}(n \cdot ???)
ans[i] = j; \leftarrow constante
```

...y finalmente lo escribimos a la salida.

```
\begin{array}{lll} \text{cout} & << j & << ' \ '; & \leftarrow \text{constante} \\ \} \end{array}
```

```
int j = i-1;
while (arr[j] >= arr[i])
    j = ans[j];
```

```
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.

```
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.

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:

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?

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}</pre>
```

• 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 << ' ';
}</pre>
```

• 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 << ' ';
}</pre>
```

- Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando i es el índice del elemento (costo constante).
- Quando 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 << ' ';
}</pre>
```

- Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando i es el índice del elemento (costo constante).
- 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 << ' ';
}</pre>
```

- Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando i es el índice del elemento (costo constante).
- elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando j es el índice del elemento (costo ????).

Quando lo comparamos con otro

```
for (int i = 1; i <= n; i++) {
   int j = i-1;
   while (arr[j] >= arr[i])
        j = ans[j];
   ans[i] = j;
   cout << j << ' ';
}</pre>
```

En otras palabras, asignamos las operaciones dentro del while al elemento arr[j], y las de fuera al elemento arr[i].

- Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando i es el índice del elemento (costo constante).
- elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando j es el índice del elemento (costo ????).

Quando lo comparamos con otro

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}</pre>
```

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!

- Cuando encontramos el primero menor a la izquierda del elemento, es decir, cuando i es el índice del elemento (costo constante).
- elemento el cual estamos buscando su primero menor a izquierda, es decir, cuando j es el índice del elemento (costo ????).

Quando lo comparamos con otro

```
for (int i = 1; i <= n; i++) {
    int j = i-1;
    while (arr[j] >= arr[i])
        j = ans[j];
    ans[i] = j;
    cout << j << ' ';
}</pre>
```

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!

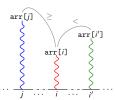
En efecto, veamos que cada elemento arr[j] sólo se lo recorrerá una sóla vez con el while (valiendo la condición) entre todas las iteraciones del for:

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.

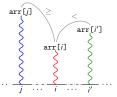
Si se recorre $\operatorname{arr}[j]$ valiendo la condición en una operación de tipo 2, es porque $\operatorname{arr}[j] \geq \operatorname{arr}[i]$ donde i se corresponde el índice del for. Así, al buscar el primero menor a izquierda de $\operatorname{arr}[i']$ con i' > i, nos encontramos con dos casos:

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).

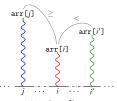
• arr[i'] > arr[i]: El primero menor a izquierda de arr[i'] es arr[i] o un número en un índice mayor a i, evitándose recorrer arr[j] en el while (recorre números de forma decreciente).



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).

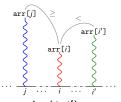


arr[i'] > arr[i]: El primero menor a izquierda de arr[i'] es arr[i] o un número en un índice mayor a i, evitándose recorrer arr[j] en el while (recorre números de forma decreciente).

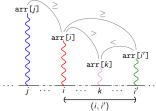


arr[i'] ≤ arr[i]: Si para algún índice k en el intervalo (i, i'), arr[k] < arr[i'], luego no se recorre en el while ningún índice menor, en particular no recorriéndose arr[j] pues j < i < k < i'.</p>

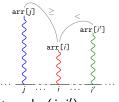
arr[i'] > arr[i]: El primero menor a izquierda de arr[i'] es arr[i] o un número en un índice mayor a i, evitándose recorrer arr[j] en el while (recorre números de forma decreciente).



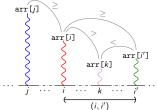
arr[i'] \leq arr[i]: Si para algún índice k en el intervalo (i, i'), arr[k] < arr[i'], luego no se recorre en el while ningún índice menor, en particular **no** recorriéndose arr[j] pues j < i < k < i'.



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).

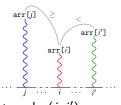


arr[i'] \leq arr[i]: Si para algún índice k en el intervalo (i, i'), arr[k] < arr[i'], luego no se recorre en el while ningún índice menor, en particular **no** recorriéndose arr[j] pues j < i < k < i'.

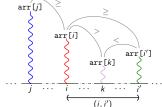


Sino, en el while se recorre a arr[i], ya que debe ser primero menor de algún elemento en (i, i').

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).



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 $arr[i] \le arr[j]$, **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.

i 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.

i 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**.

i 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.

- Introducción
- 2 Complejidad computaciona
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

- Introducciór
- 2 Complejidad computaciona
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

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.

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.

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...

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.

- Introducción
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

omegaUp

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 Torneo Jujeño de Programación, por ejemplo.

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 Torneo Jujeño de Programación, por ejemplo.

También es utilizado para desarrollar de forma virtual todos los años la OII (Olimpíada Iberoamericana de Informática).

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 Torneo Jujeño de Programación, por ejemplo.

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 preparen a futuro.

Login / crear una cuenta

¡Veámoslo en vivo!

Página principal

omegaUp Concursos ▼ Cursos ▼ Problemas ▼ Ranking ▼ Ayuda ▼

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".





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.

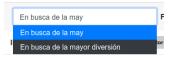
A la izquierda de cada problema aparece su id. Es muy común en los jueces online que cada problema tenga su id único.

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.











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).



- 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).



- 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...



- 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.



- 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.

El formato de un problema depende de la competencia original donde se usó (si la hay) y/o del juez.

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:

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:

Título / nombre del 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 🤶

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):

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 🤉

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.

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 🤉

- 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*.

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 🤅

- 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.

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		

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):

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.

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.

ONTITION DE CONTRE LA CON

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.

Solution 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.

Solution 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:



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.

Solution 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:



Es decir, en este problema, al correr cada caso nuestra solución puede reservar a lo sumo 4.8828125 MiB. Número raro...

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 consequir la mayor diversión posible.

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.

Entrada:

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.

Entrada: Suele incluir detalles específicos sobre cómo se provee la información del problema al programa que implementemos.

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 consequir la mayor diversión posible.

© 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.

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.

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

- \bullet Una línea con la cantidad N de juguetes que eligió inicialmente Nicolás.
- ullet N líneas, cada una con el nivel de diversión de un juguete.



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). 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

Salida

"no existe" (según el problema).

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).

solución imprimir" si es que hay varias o qué imprimir si una solución

Ejemplo(s):

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).

☑ 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.

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).

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.

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).

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.



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*.



• Cotas: Establecen el intervalo o conjunto de valores que toman los valores de la entrada y/o la cantidad máxima de elementos que pueden haber en un caso de prueba.

Son muy importantes junto con el límite de tiempo en cuanto a *qué* tan eficiente debe ser una solución para resolver el problema.

Cotas: Establecen el intervalo o conjunto de valores que toman los valores de la entrada y/o la cantidad máxima de elementos que pueden haber en un caso de prueba. Son muy importantes junto con el límite de tiempo en cuanto a qué tan eficiente debe ser una solución para resolver el problema. En efecto, según la cota asintótica superior de la solución, la cantidad de operaciones que realiza respecto a la entrada puede ser o no suficiente para correr en el tiempo límite establecido para el problema,

y nuestra elección depende de qué sea "más rápido" (como cuando

discutimos elegir selection vs merge sort).

• Cotas: Establecen el intervalo o conjunto de valores que toman los valores de la entrada y/o la cantidad máxima de elementos que pueden haber en un caso de prueba.
Son muy importantes junto con el límite de tiempo en cuanto a qué tan eficiente debe ser una solución para resolver el problema.
En efecto, según la cota asintótica superior de la solución, la cantidad de operaciones que realiza respecto a la entrada puede ser o no suficiente para correr en el tiempo límite establecido para el problema, y nuestra elección depende de qué sea "más rápido" (como cuando discutimos elegir selection vs merge sort).

Cotas

- 2 < N < 1.000.000
- $1 \le Nivel\ de\ diversion \le 100$

En efecto, según la cota asintótica superior de la solución, la cantidad de operaciones que realiza respecto a la entrada puede ser o no suficiente para correr en el tiempo límite establecido para el problema, y nuestra elección depende de qué sea "más rápido" (como cuando discutimos elegir selection vs merge sort).

Cotas

- $2 \le N \le 1.000.000$
- $1 \le Nivel\ de\ diversion \le 100$

En este caso, cada caso de prueba tiene a lo sumo $1,000,000=10^6$ números, cada uno con un valor entre $1\ y\ 100.$

Subtareas: Sólo existen en las competencias tipo "IOI" donde cada problema otorga hasta 100 puntos. Así, no hay subtareas en ICPC. ■ Subtareas: Sólo existen en las competencias tipo "IOI" donde cada problema otorga hasta 100 puntos. Así, no hay subtareas en ICPC. Establecen restricciones adicionales al enunciado original de forma que si se sube una solución correcta bajo la restricción agregada, otorgan puntaje parcial, una cantidad menor a 100 puntos indicado al lado de la subtarea.

▼ Subtareas: Sólo existen en las competencias tipo "IOI" donde cada problema otorga hasta 100 puntos. Así, no hay subtareas en ICPC. Establecen restricciones adicionales al enunciado original de forma que si se sube una solución correcta bajo la restricción agregada, otorgan puntaje parcial, una cantidad menor a 100 puntos indicado al lado de la subtarea.

Subtareas

- N=2 (19 puntos)
- $N \leq 1.000$ (30 puntos)
- $N \le 40.000$ (33 puntos)
- Sin más restricciones (18 puntos)

Subtareas

- N=2 (19 puntos)
- $N \le 1.000$ (30 puntos)
- $N \le 40.000$ (33 puntos)
- Sin más restricciones (18 puntos)

Por ejemplo, para este problema, si lo resolvemos suponiendo que se dan sólo dos números en la entrada obtenemos 19 puntos.

la subtarea.

Subtareas: Sólo existen en las competencias tipo "IOI" donde cada problema otorga hasta 100 puntos. Así, no hay subtareas en ICPC. Establecen restricciones adicionales al enunciado original de forma que si se sube una solución correcta bajo la restricción agregada, otorgan puntaje parcial, una cantidad menor a 100 puntos indicado al lado de la subtarea.

Subtareas

- N=2 (19 puntos)
- $N \leq 1.000$ (30 puntos)
- $N \le 40.000$ (33 puntos)
- Sin más restricciones (18 puntos)

Por ejemplo, para este problema, si lo resolvemos suponiendo que se dan sólo dos números en la entrada obtenemos 19 puntos. De 100 puntos, 19 por algo sencillo no está mal, no?

▼ Subtareas: Sólo existen en las competencias tipo "IOI" donde cada problema otorga hasta 100 puntos. Así, no hay subtareas en ICPC. Establecen restricciones adicionales al enunciado original de forma que si se sube una solución correcta bajo la restricción agregada, otorgan puntaje parcial, una cantidad menor a 100 puntos indicado al lado de la subtarea.

Subtareas

- N=2 (19 puntos)
- $N \leq 1.000$ (30 puntos)
- $N \leq 40.000$ (33 puntos)
- Sin más restricciones (18 puntos)

Por ejemplo, para este problema, si lo resolvemos suponiendo que se dan **sólo dos números en la entrada** obtenemos 19 puntos. De 100 puntos, 19 por algo sencillo no está mal, no?

"Sin más restricciones" indica los puntos otorgados por encima de las otras subtareas, es decir, puntos que sólo podríamos obtener si se resuelve el problema correctamente sin restricciones adicionales al enunciado original, como dice el nombre.

Subtareas (cont.): En competencias tipo IOI es muy importante prestar atención a las subtareas, porque nos permiten obtener puntaje sin encontrar la solución del problema sin restricciones, que puede ser significativamente más difícil de pensar e incluso implementar.

- Subtareas (cont.): En competencias tipo IOI es muy importante prestar atención a las subtareas, porque nos permiten obtener puntaje sin encontrar la solución del problema sin restricciones, que puede ser significativamente más difícil de pensar e incluso implementar.
 - La mayoría de jueces suelen contener más problemas tipo ICPC, es decir, que no incluyen subtareas. Más adelante mencionaremos qué jueces y en dónde contienen problemas de este tipo.

Subtareas (cont.): En competencias tipo IOI es muy importante prestar atención a las subtareas, porque nos permiten obtener puntaje sin encontrar la solución del problema sin restricciones, que puede ser significativamente más difícil de pensar e incluso implementar.

La mayoría de jueces suelen contener más problemas tipo ICPC, es decir, que no incluyen subtareas. Más adelante mencionaremos qué jueces y en dónde contienen problemas de este tipo.

Una última consideración a tener en cuenta parte de cómo se suman los puntajes de las subtareas.

Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:

- Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:
 - Una forma es que se puedan subir múltiples soluciones a un problema, y que se sumen los mejores puntajes por subtarea entre las distintas soluciones subidas

- Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:
 - Una forma es que se puedan subir múltiples soluciones a un problema, y que se sumen los mejores puntajes por subtarea entre las distintas soluciones subidas (por ejemplo, selectivo de OIA e IOI).

- Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:
 - Una forma es que se puedan subir múltiples soluciones a un problema, y que se sumen los mejores puntajes por subtarea entre las distintas soluciones subidas (por ejemplo, selectivo de OIA e IOI).
 - Otra forma más clásica es que simplemente se tome el mayor puntaje entre todas las subidas, y que sea responsabilidad del competidor detectar la subtarea en función de la entrada en el código, a fin de ejecutar la solución correspondiente a la subtarea que hayamos pensado, y sumar los puntos respectivos

- Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:
 - Una forma es que se puedan subir múltiples soluciones a un problema, y que se sumen los mejores puntajes por subtarea entre las distintas soluciones subidas (por ejemplo, selectivo de OIA e IOI).
 - Otra forma más clásica es que simplemente se tome el mayor puntaje entre todas las subidas, y que sea responsabilidad del competidor detectar la subtarea en función de la entrada en el código, a fin de ejecutar la solución correspondiente a la subtarea que hayamos pensado, y sumar los puntos respectivos (por ejemplo, omegaUp, y en particular, el Torneo Jujeño de Programación).

- Subtareas (cont.): Hay dos formas usuales en las que se maneja la suma de los puntajes de las subtareas en las competencias y jueces online:
 - Una forma es que se puedan subir múltiples soluciones a un problema, y que se sumen los mejores puntajes por subtarea entre las distintas soluciones subidas (por ejemplo, selectivo de OIA e IOI).
 - Otra forma más clásica es que simplemente se tome el mayor puntaje entre todas las subidas, y que sea responsabilidad del competidor detectar la subtarea en función de la entrada en el código, a fin de ejecutar la solución correspondiente a la subtarea que hayamos pensado, y sumar los puntos respectivos (por ejemplo, omegaUp, y en particular, el Torneo Jujeño de Programación).

Idealmente deben consultar el reglamento de la competencia y el funcionamiento del juez en cada caso.

- Introducción
- 2 Complejidad computaciona
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

Recordemos el enunciado y la solución propuesta al problema de antes:

Recordemos el enunciado y la solución propuesta al problema de antes:

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.

Recordemos el enunciado y la solución propuesta al problema de antes:

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 Njuguetes, dados por entrada. Queremos obtener la mayor diversión posible con N-1 juguetes de los N dados.

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

Recordemos el enunciado y la solución propuesta al problema de antes:

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 Njuguetes, dados por entrada. Queremos obtener la mayor diversión posible con N-1 juguetes de los N dados.

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

¡Probemos subirlo al juez!

Veredictos

曲 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	S Memoria	Tiempo	Acciones
2025-05-30 20:03	cpp11-gcc	82.00%	Interrumpida	Interrumpida 1	Límite excedido	3.32 s	Q

曲 Fecha y hora	a Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:03	cpp11-gcc	82.00%	Interrumpida	Interrumpida	Límite excedido	3.32 s	(Q)

Nuestro primer veredicto!

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	S Memoria	Tiempo	Acciones
2025-05-30 20:03	cpp11-gcc	82.00%	Interrumpida	Interrumpida	Límite excedido	3.32 s	Q

Nuestro primer veredicto! Sacamos 82ptos (82 % del total en omegaUp), pero nos excedimos del límite de memoria.

描 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	S Memoria	Tiempo	Acciones
2025-05-30 20:03	cpp11-gcc	82.00%	Interrumpida	Interrumpida	Límite excedido	3.32 s	(Q)

Nuestro primer veredicto! Sacamos 82ptos (82 % del total en omegaUp), pero nos excedimos del límite de memoria.

Si recordamos las subtareas, significa que obtuvimos *puntaje parcial*, sumando puntos de todas las subtareas menos la última "sin restricciones":

Subtareas

- N=2 (19 puntos)
- N < 1.000 (30 puntos)
- N < 40.000 (33 puntos)
- Sin más restricciones (18 puntos)

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:03	cpp11-gcc	82.00%	Interrumpida	Interrumpida	Límite excedido	3.32 s	Q

Nuestro primer veredicto! Sacamos 82ptos (82 % del total en omegaUp), pero nos excedimos del límite de memoria.

Si recordamos las subtareas, significa que obtuvimos *puntaje parcial*, sumando puntos de todas las subtareas menos la última "sin restricciones":

Subtareas

- N = 2 (19 puntos)
- $N \leq 1.000$ (30 puntos)
- $N \leq 40.000$ (33 puntos)
- Sin más restricciones (18 puntos)

Es decir, nuestro código es correcto para N=2, para $N\leq 1000$ y para $N\leq 40000$.

Se acuerdan que habíamos dicho que el problema tiene trampa?

Se acuerdan que habíamos dicho que el problema tiene trampa? Bueno, acá está: el límite de memoria es muy bajo.

Se acuerdan que habíamos dicho que el problema tiene trampa? Bueno, acá está: el límite de memoria es muy bajo. Esto explica un poco el número raro que habíamos visto...

Se acuerdan que habíamos dicho que el problema tiene trampa? Bueno, acá está: el límite de memoria es muy bajo. Esto explica un poco el número raro que habíamos visto...

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		

Se acuerdan que habíamos dicho que el problema tiene trampa? Bueno, acá está: el límite de memoria es muy bajo. Esto explica un poco el número raro que habíamos visto...

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		

Si bien nos enfocamos en esta clase en analizar cuánto tiempo tarda en correr una solución, podemos ver brevemente por qué nos pasamos del límite de memoria.

• Observemos que en la solución creamos un vector<int> de N elementos.

• Observemos que en la solución creamos un vector<int> de N elementos. Supongamos que nuestra solución ejecuta un caso con $N=10^6$ números, la máxima posible según las cotas del problema (y una cantidad mayor a las subtareas donde sabemos que la solución es correcta).

- Observemos que en la solución creamos un vector<int> de N elementos. Supongamos que nuestra solución ejecuta un caso con $N=10^6$ números, la máxima posible según las cotas del problema (y una cantidad mayor a las subtareas donde sabemos que la solución es correcta).
- Entonces, como cada int ocupa 32 bits (4 bytes), un vector<int> de $N=10^6$ elementos ocupa alrededor de $4\cdot 10^6$ bytes, o sea, $4\cdot 10^6/1024\approx 3906$ KiB (kibibytes), que son $3906/1024\approx 3,8147$ MiB (mebibytes).

- Observemos que en la solución creamos un vector<int> de N elementos. Supongamos que nuestra solución ejecuta un caso con $N=10^6$ números, la máxima posible según las cotas del problema (y una cantidad mayor a las subtareas donde sabemos que la solución es correcta).
- Entonces, como cada int ocupa 32 bits (4 bytes), un vector<int> de $N=10^6$ elementos ocupa alrededor de $4\cdot 10^6$ bytes, o sea, $4\cdot 10^6/1024\approx 3906$ KiB (kibibytes), que son $3906/1024\approx 3,8147$ MiB (mebibytes).
- Si bien 3,8147 MiB < 4,8828125 MiB, no son números muy lejanos entre sí, y un programa puede reservar algo de memoria adicional para funcionar correctamente (y también reservamos algunos int más, aunque el espacio que ocupan no es significante).

- Observemos que en la solución creamos un vector<int> de N elementos. Supongamos que nuestra solución ejecuta un caso con $N=10^6$ números, la máxima posible según las cotas del problema (y una cantidad mayor a las subtareas donde sabemos que la solución es correcta).
- Entonces, como cada int ocupa 32 bits (4 bytes), un vector<int> de $N=10^6$ elementos ocupa alrededor de $4\cdot 10^6$ bytes, o sea. $4 \cdot 10^6 / 1024 \approx 3906 \text{ KiB (kibibytes)}$, que son $3906 / 1024 \approx 3,8147 \text{ MiB}$ (mebibytes).
- Si bien 3,8147 MiB < 4,8828125 MiB, no son números muy lejanos entre sí, y un programa puede reservar algo de memoria adicional para funcionar correctamente (y también reservamos algunos int más, aunque el espacio que ocupan no es significante).

Nos preguntamos entonces: ¿Existe una forma de usar menos memoria?

Observación

Observación

Cada número de la entrada está entre 1 y 100 (recordar las cotas).

Observación

Cada número de la entrada está entre 1 y 100 (recordar las cotas). Como un byte puede almacenar números en el intervalo [-128,127], cada número entra en un byte.

Observación

Cada número de la entrada está entre 1 y 100 (recordar las cotas). Como un byte puede almacenar números en el intervalo [-128, 127], cada número entra en un byte. Entonces, podemos reemplazar nuestro vector < int > con un <math>vector < char > !

Observación

Cada número de la entrada está entre 1 y 100 (recordar las cotas). Como un byte puede almacenar números en el intervalo [-128, 127], cada número entra en un byte. Entonces, podemos reemplazar nuestro vector < int > con un <math>vector < char > !

Así, teniendo cuidado de hacer casts a int cuando sea necesario:

Observación

Cada número de la entrada está entre 1 y 100 (recordar las cotas). Como un byte puede almacenar números en el intervalo [-128, 127], cada número entra en un byte. Entonces, podemos reemplazar nuestro vector<int> con un vector<char>!

```
Así, teniendo cuidado de hacer casts a int cuando sea necesario:
```

```
int n: cin >> n:
vector<char> arr(n):
for (int i = 0; i < n; i++) {int x; cin >> x; arr[i] = x;}
int sum = 0, mini = arr[0];
for (int i = 0; i < n; i++)
    sum += (int)arr[i], mini = min(mini, (int)arr[i]);
cout << sum - mini << '\n':</pre>
```

¿Probamos de nuevo?



Jueces Online Veredictos



苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta	4.10 MB	3.36 s	Q





曲 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	срр11-дсс	100.00%	Terminada	Correcta ⊘	4.10 MB	3.36 s	Q

100 ptos!



苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	срр11-дсс	100.00%	Terminada	Correcta	4.10 MB	3.36 s	Q





m Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta ⊘	4.10 MB	3.36 s	Q

Por otro lado, el veredicto que tuvimos inicialmente suele recibir el nombre de *límite de memoria excedida*, o en inglés, **Memory Limit Exceeded**, abreviado normalmente como **MLE**.





Por otro lado, el veredicto que tuvimos inicialmente suele recibir el nombre de *límite de memoria excedida*, o en inglés, **Memory Limit Exceeded**, abreviado normalmente como **MLE**.

Este veredicto no es tan común en IOI o OIA, porque el límite de memoria suele ser bastante alto, típicamente (aunque con excepciones) de 512 MiB, bastante más que los \approx 4,8 MiB de este problema.

曲 Fecha y hora

2025-05-30

Lenguaje

cpp11-qcc





苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta ⊘	4.10 MB	3.36 s	@

Por otro lado, el veredicto que tuvimos inicialmente suele recibir el nombre de *límite de memoria excedida*, o en inglés, **Memory Limit Exceeded**, abreviado normalmente como **MLE**.

Este veredicto no es tan común en IOI o OIA, porque el límite de memoria suele ser bastante alto, típicamente (aunque con excepciones) de 512 MiB, bastante más que los \approx 4,8 MiB de este problema.

Más aún, no es demasiado común que el límite de memoria sea un factor importante en la solución del problema, muchas veces siendo más relevante el límite de tiempo.





苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	срр11-дсс	100.00%	Terminada	Correcta	4.10 MB	3.36 s	Q

Por otro lado, el veredicto que tuvimos inicialmente suele recibir el nombre de *límite de memoria excedida*, o en inglés, **Memory Limit Exceeded**, abreviado normalmente como **MLE**.

Este veredicto no es tan común en IOI o OIA, porque el límite de memoria suele ser bastante alto, típicamente (aunque con excepciones) de 512 MiB, bastante más que los \approx 4,8 MiB de este problema.

Más aún, no es demasiado común que el límite de memoria sea un factor importante en la solución del problema, muchas veces siendo más relevante el límite de tiempo. Es más fácil "alcanzar primero" el límite de tiempo procesando tanta información, que el de memoria.

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta	4.10 MB	3.36 s	@

Otra cosa importante a remarcar es el uso de memoria de nuestro programa: 4,10 MiB.

Jueces Online Veredictos

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	■ Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta 🕏	4.10 MB	3.36 s	(Q)

Otra cosa importante a remarcar es el uso de memoria de nuestro programa: 4,10 MiB. Si bien calculando, nuestra solución debería reservar alrededor de $1\cdot 10^6$ bytes para el vector<char> que usamos, o sea, $\approx 0,95$ MiB, la cantidad reportada por omegaUp es significativamente más alta.

Jueces Online Veredictos

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	■ Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta	4.10 MB	3.36 s	Q

Otra cosa importante a remarcar es el uso de memoria de nuestro programa: 4,10 MiB. Si bien calculando, nuestra solución debería reservar alrededor de $1\cdot 10^6$ bytes para el vector<char> que usamos, o sea, $\approx 0,95$ MiB, la cantidad reportada por omegaUp es significativamente más alta. Esto puede ser por diversos factores, como que std::vector almacena otro tipo de información que ocupa memoria, entre otros elementos del funcionamiento de nuestro programa en C++ que requieren reservar más memoria de la esperada.

Jueces Online Veredictos

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	■ Memoria	Tiempo	Acciones
2025-05-30 20:32	cpp11-gcc	100.00%	Terminada	Correcta	4.10 MB	3.36 s	Q

Otra cosa importante a remarcar es el uso de memoria de nuestro programa: 4,10 MiB. Si bien calculando, nuestra solución debería reservar alrededor de $1\cdot 10^6$ bytes para el vector<char> que usamos, o sea, $\approx 0,95$ MiB, la cantidad reportada por omegaUp es significativamente más alta. Esto puede ser por diversos factores, como que std::vector almacena otro tipo de información que ocupa memoria, entre otros elementos del funcionamiento de nuestro programa en C++ que requieren reservar más memoria de la esperada.

Por este motivo, si obtenemos un veredicto **MLE**, debemos ser generosos en nuestra estimación de memoria utilizada, con el fin de tener cierto margen para que nuestra solución respete el límite de memoria del problema.

Observación

Se pueden procesar los números sin almacenarlos en un std::vector.

Observación

Se pueden procesar los números sin almacenarlos en un std::vector.

En efecto, como sólo queremos obtener la suma y el mínimo del arreglo, directamente podemos procesarlos a medida se los lee desde la entrada, sin crear un arreglo para almacenar los enteros:

Observación

Se pueden procesar los números sin almacenarlos en un std::vector.

En efecto, como sólo queremos obtener la suma y el mínimo del arreglo, directamente podemos procesarlos a medida se los lee desde la entrada, sin crear un arreglo para almacenar los enteros:

```
int n; cin >> n;
int sum = 0:
int mini = 150; // 150 es más grande que cualquier elem. de la entrada
for (int i = 0; i < n; i++) {
    int x; cin >> x;
    sum += x, mini = min(mini, x);
cout << sum - mini << '\n':</pre>
```

Observación

Se pueden procesar los números sin almacenarlos en un std::vector.

En efecto, como sólo queremos obtener la suma y el mínimo del arreglo, directamente podemos procesarlos a medida se los lee desde la entrada, sin crear un arreglo para almacenar los enteros:

```
int n; cin >> n;
int sum = 0:
int mini = 150; // 150 es más grande que cualquier elem. de la entrada
for (int i = 0; i < n; i++) {
    int x; cin >> x;
    sum += x, mini = min(mini, x);
cout << sum - mini << '\n':</pre>
```

Cuando lo subimos al juez la solución es correcta y consume aún menos memoria:

Observación

Se pueden procesar los números sin almacenarlos en un std::vector.

En efecto, como sólo queremos obtener la suma y el mínimo del arreglo, directamente podemos procesarlos a medida se los lee desde la entrada, sin crear un arreglo para almacenar los enteros:

```
int n; cin >> n;
int sum = 0:
int mini = 150; // 150 es más grande que cualquier elem. de la entrada
for (int i = 0; i < n; i++) {
    int x; cin >> x;
    sum += x, mini = min(mini, x);
cout << sum - mini << '\n':</pre>
```

Cuando lo subimos al juez la solución es correcta y consume aún menos memoria:

苗 Fecha y hora	Lenguaje	Porcentaje	Ejecución	Salida	S Memoria	Tiempo	Acciones
2025-05-30 21:30	cpp11-gcc	100.00%	Terminada	Correcta	3.38 MB	3.39 s	Q

El problema de antes no es el único que introdujimos en la sección de complejidad:

El problema de antes no es el único que introdujimos en la sección de complejidad:

Problema (Watermelon, CodeForces 4A)

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

El problema de antes no es el único que introdujimos en la sección de complejidad:

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?

El juez es otro ahora:

El problema de antes no es el único que introdujimos en la sección de complejidad:

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?

El juez es otro ahora:



El problema de antes no es el único que introdujimos en la sección de complejidad:

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?

El juez es otro ahora:



Es uno de los jueces más populares, con muchos problemas (tipo ICPC), competencias de alta calidad organizadas de forma regular y un sistema de rating.

El problema de antes no es el único que introdujimos en la sección de complejidad:

Problema (Watermelon, CodeForces 4A)

Pete y Billy quieren dividir una sandía de peso w kg en dos pedazos con peso par. Dado w_i jes posible?

El juez es otro ahora:



Es uno de los jueces más populares, con muchos problemas (tipo ICPC), competencias de alta calidad organizadas de forma regular y un sistema de rating.

¡Busquemos el problema en el juez!

• TL: 1 segundo

• TL: 1 segundo

ML: 64MB (acá son megabytes, no mebibytes)

- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.

- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.
- Entrada y Cotas: Un entero w con $1 \le w \le 100$.

- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.
- Entrada y Cotas: Un entero w con $1 \le w \le 100$.
- Salida: Imprimir YES o NO según si se puede dividir la sandía.

- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.
- Entrada y Cotas: Un entero w con $1 \le w \le 100$.
- Salida: Imprimir YES o NO según si se puede dividir la sandía.
- Ejemplo:



- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.
- Entrada y Cotas: Un entero w con $1 \le w \le 100$.
- Salida: Imprimir YES o NO según si se puede dividir la sandía.
- Ejemplo:



Una explicación del ejemplo está en la sección *Note*:

Note

For example, the boys can divide the watermelon into two parts of 2 and 6 kilos respectively (another variant — two parts of 4 and 4 kilos).

- TL: 1 segundo
- ML: 64MB (acá son megabytes, no mebibytes)
- Enunciado: Ya lo sabemos, la dada en la diapositiva anterior es una versión digerida del original.
- Entrada y Cotas: Un entero w con $1 \le w \le 100$.
- Salida: Imprimir YES o NO según si se puede dividir la sandía.
- Ejemplo:



Una explicación del ejemplo está en la sección Note:

Note

For example, the boys can divide the watermelon into two parts of 2 and 6 kilos respectively (another variant — two parts of 4 and 4 kilos).

Subtareas: No hay, es estilo ICPC.

Recordemos la solución que propusimos al problema:

Recordemos la solución que propusimos al problema:

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

Recordemos la solución que propusimos al problema:

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

¡Probemos subirlo al juez!

Recordemos la solución que propusimos al problema:

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

¡Probemos subirlo al juez!

#	When	Who	Problem	Lang	Verdict	Time	Memory
322115237	May/30/2025 23:36 ^{UTC-3}	ariloc	4A - Watermelon	C++17 (GCC 7-32)	Wrong answer on test 5	92 ms	100 KB

Recordemos la solución que propusimos al problema:

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

¡Probemos subirlo al juez!

#	When	Who	Problem	Lang	Verdict	Time	Memory
322115237	May/30/2025 23:36 ^{UTC-3}	ariloc	4A - Watermelon	C++17 (GCC 7-32)	Wrong answer on test 5	92 ms	100 KB

Obtuvimos un nuevo veredicto: Wrong Answer (abreviado como WA), es decir, *respuesta incorrecta*.

Recordemos la solución que propusimos al problema:

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

¡Probemos subirlo al juez!



Obtuvimos un nuevo veredicto: Wrong Answer (abreviado como WA), es decir, *respuesta incorrecta*.

Esto significa que nuestra solución es incorrecta, pero no porque hayamos excedido algún límite, sino porque nuestra solución en algún caso de prueba imprime una respuesta que **no es correcta** según el enunciado.

• La solución que pensamos al problema es incorrecta, no lo resuelve siempre.

- La solución que pensamos al problema es incorrecta, no lo resuelve siempre.
- Nuestra implementación de la solución tiene un error, un bug, que hace que no represente la idea que queremos reflejar en código.

- La solución que pensamos al problema es incorrecta, no lo resuelve siempre.
- Nuestra implementación de la solución tiene un error, un bug, que hace que no represente la idea que queremos reflejar en código.

Por supuesto, hay variantes a estos motivos. Hay casos donde la idea está "casi bien", y capaz nos falta considerar un caso muy específico en nuestro código, por ejemplo para entradas muy chicas (n = 0, n = 1, etc, si el problema lo permite). A este tipo de casos los conocemos como casos borde.

- La solución que pensamos al problema es incorrecta, no lo resuelve siempre.
- Nuestra implementación de la solución tiene un error, un bug, que hace que no represente la idea que queremos reflejar en código.

Por supuesto, hay variantes a estos motivos. Hay casos donde la idea está "casi bien", y capaz nos falta considerar un caso muy específico en nuestro código, por ejemplo para entradas muy chicas (n = 0, n = 1, etc, si el problema lo permite). A este tipo de casos los conocemos como casos borde.

Curiosamente, es precisamente un caso borde lo que hace que nuestra solución al problema anterior falle.

Pensemos por un momento: ¿basta sólo con que el peso de la sandía sea par para poder dividirla en dos pedazos pares?

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

En efecto, sabemos que $1 \le w \le 100$, entonces w = 2 es una posible entrada a nuestro programa:

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

En efecto, sabemos que $1 \le w \le 100$, entonces w = 2 es una posible entrada a nuestro programa:

• w = 2 es par, entonces nuestra solución anterior imprimiría que la sandía sí se puede dividir en dos pedazos pares.

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

En efecto, sabemos que $1 \le w \le 100$, entonces w = 2 es una posible entrada a nuestro programa:

- w = 2 es par, entonces nuestra solución anterior imprimiría que la sandía sí se puede dividir en dos pedazos pares.
- Sin embargo, la única forma de dividir la sandía en dos partes con peso entero es que ambas pesen 1kg, por lo que no se la puede dividir en pedazos de peso par.

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

En efecto, sabemos que $1 \le w \le 100$, entonces w = 2 es una posible entrada a nuestro programa:

- w = 2 es par, entonces nuestra solución anterior imprimiría que la sandía sí se puede dividir en dos pedazos pares.
- Sin embargo, la única forma de dividir la sandía en dos partes con peso entero es que ambas pesen 1kg, por lo que no se la puede dividir en pedazos de peso par.

Encontramos nuestro caso borde!

Dedujimos esto porque vimos que si es posible, como la suma de dos núemros pares es par, entonces w debe ser par. Pero esto no implica que sea al revés.

En efecto, sabemos que $1 \le w \le 100$, entonces w = 2 es una posible entrada a nuestro programa:

- w = 2 es par, entonces nuestra solución anterior imprimiría que la sandía sí se puede dividir en dos pedazos pares.
- Sin embargo, la única forma de dividir la sandía en dos partes con peso entero es que ambas pesen 1kg, por lo que no se la puede dividir en pedazos de peso par.

Encontramos nuestro caso borde! ¿Pero cómo lo resolvemos? ¿Es el único caso donde nuestra solución estaba mal?

Analicemos lo que sabemos:

Analicemos lo que sabemos:

• w tiene que ser par para que la respuesta sea YES.

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w > 2 y w par, la respuesta es siempre YES.

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w>2 y w par, la respuesta es siempre YES. Esto se debe a que, si w cumple las condiciones, en particular $w\geq 4$, entonces se puede dividir a w en un pedazo de peso 2kg, y otro de peso (w-2)kg, que será positivo y par porque restamos dos números pares.

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w>2 y w par, la respuesta es siempre YES. Esto se debe a que, si w cumple las condiciones, en particular $w\geq 4$, entonces se puede dividir a w en un pedazo de peso 2kg, y otro de peso (w-2)kg, que será positivo y par porque restamos dos números pares. De esta forma, sólo debemos cubrir el caso borde mencionado en la solución antes presentada:

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w>2 y w par, la respuesta es siempre YES. Esto se debe a que, si w cumple las condiciones, en particular $w\geq 4$, entonces se puede dividir a w en un pedazo de peso $2 \log y$, y otro de peso $(w-2) \log y$, que será positivo y par porque restamos dos números pares. De esta forma, sólo debemos cubrir el caso borde mencionado en la solución antes presentada:

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

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w>2 y w par, la respuesta es siempre YES. Esto se debe a que, si w cumple las condiciones, en particular $w\geq 4$, entonces se puede dividir a w en un pedazo de peso $2 \log y$, y otro de peso $(w-2) \log y$, que será positivo y par porque restamos dos números pares. De esta forma, sólo debemos cubrir el caso borde mencionado en la solución antes presentada:

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

¿Probamos de nuevo?

Analicemos lo que sabemos:

- w tiene que ser par para que la respuesta sea YES.
- Si w = 2, la respuesta es NO.

Así, veamos que si w>2 y w par, la respuesta es siempre YES. Esto se debe a que, si w cumple las condiciones, en particular $w\geq 4$, entonces se puede dividir a w en un pedazo de peso 2kg, y otro de peso (w-2)kg, que será positivo y par porque restamos dos números pares. De esta forma, sólo debemos cubrir el caso borde mencionado en la solución antes presentada:

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

¿Probamos de nuevo?

When Who Problem Lang Verdict Time Memory 322118155 May/31/2025 00:24^{MC-3} ariloc 4A - Watermelon C++17 (GCC 7-32) Accepted 92 ms 100 KB

Tanto decir que es importante que nuestro programa vaya rápido, veamos ahora cómo el juez nos informa que nuestro programa excedió el *límite de tiempo del problema*. Para ello, aprovechemos otro problema más que introdujimos antes:

Tanto decir que es importante que nuestro programa vaya rápido, veamos ahora cómo el juez nos informa que nuestro programa excedió el *límite de tiempo del problema*. Para ello, aprovechemos otro problema más que introdujimos antes:

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. $1 < n < 2 \cdot 10^5$

Tanto decir que es importante que nuestro programa vaya rápido, veamos ahora cómo el juez nos informa que nuestro programa excedió el *límite de tiempo del problema*. Para ello, aprovechemos otro problema más que introdujimos antes:

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. $1 < n < 2 \cdot 10^5$

Otro juez más:

Tanto decir que es importante que nuestro programa vaya rápido, veamos ahora cómo el juez nos informa que nuestro programa excedió el *límite de tiempo del problema*. Para ello, aprovechemos otro problema más que introdujimos antes:

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. $1 < n < 2 \cdot 10^5$

Otro juez más:







Es un juez utilizado en Finlandia para organizar competencias que tiene un conjunto de problemas reducido pero de muy buena calidad, orientados a educar en diversos temas de Programación Competitiva.



Es un juez utilizado en Finlandia para organizar competencias que tiene un conjunto de problemas reducido pero de muy buena calidad, orientados a educar en diversos temas de Programación Competitiva.

Uno de sus principales desarrolladores es Antti Laaksonen, escritor del libro "Guide To Competitive Programming". Muchos de los problemas y soluciones que describe en el libro pueden resolverse en CSES.



Es un juez utilizado en Finlandia para organizar competencias que tiene un conjunto de problemas reducido pero de muy buena calidad, orientados a educar en diversos temas de Programación Competitiva.

Uno de sus principales desarrolladores es Antti Laaksonen, escritor del libro "Guide To Competitive Programming". Muchos de los problemas y soluciones que describe en el libro pueden resolverse en CSES.

¡Busquemos el problema en el juez!

No es problema identificar las partes del problema en CSES.

No es problema identificar las partes del problema en CSES. Lo que más nos interesa es que el TL es 1s, y el ML es 512MB, números muy comunes para problemas de Programación Competitiva.

No es problema identificar las partes del problema en CSES. Lo que más nos interesa es que el TL es 1s, y el ML es 512MB, números muy comunes para problemas de Programación Competitiva. Además no hay subtareas, es estilo ICPC.

Además no hay subtareas, es estilo ICPC.

No es problema identificar las partes del problema en CSES. Lo que más nos interesa es que el TL es 1s, y el ML es 512MB, números muy comunes para problemas de Programación Competitiva.

En la sección de análisis amortizado (si no la salteamos por tiempo) vimos una solución en $\mathcal{O}(n)$ que resuelve este problema. Dicha solución es correcta y mandarla al juez resulta en un veredicto Accepted, se los prometo.

No es problema identificar las partes del problema en CSES.

Lo que más nos interesa es que el TL es 1s, y el ML es 512MB, números muy comunes para problemas de Programación Competitiva. Además no hay subtareas, es estilo ICPC.

En la sección de análisis amortizado (si no la salteamos por tiempo) vimos una solución en $\mathcal{O}(n)$ que resuelve este problema. Dicha solución es correcta y mandarla al juez resulta en un veredicto Accepted, se los prometo.

Lo que nos vamos a enfocar es ver qué pasa si mandamos una solución que es muy lenta, como una que corre en $\mathcal{O}(n^2)$.

Proponemos la siguiente solución "ingenua" o *naive* al problema, que corre en $\mathcal{O}(n^2)$:

Proponemos la siguiente solución "ingenua" o naive al problema, que corre en $\mathcal{O}(n^2)$:

Proponemos la siguiente solución "ingenua" o *naive* al problema, que corre en $\mathcal{O}(n^2)$:

Siendo $1 \le n \le 2 \cdot 10^5$, la cantidad de operaciones que ejecuta una solución con una cota asintótica tal supera ampliamente el rule of thumb que conocemos para que el programa termine en 1s.

Proponemos la siguiente solución "ingenua" o *naive* al problema, que corre en $\mathcal{O}(n^2)$:

Siendo $1 \le n \le 2 \cdot 10^5$, la cantidad de operaciones que ejecuta una solución con una cota asintótica tal supera ampliamente el rule of thumb que conocemos para que el programa termine en 1s.

¡Probemos subirlo al juez!

Submission details

Task:	Nearest Smaller Values	
Sender:	ariloc	
Submission time:	2025-05-31 07:06:22 +0300	
Language:	C++ (C++20)	
Status:	READY	
Result:	TIME LIMIT EXCEEDED	

Test results

test	verdict	time	
#1	ACCEPTED	0.00 s	<u>>></u>
#2	ACCEPTED	0.00 s	<u>>></u>
#3	ACCEPTED	0.00 s	<u>>></u>
#4	ACCEPTED	0.04 s	<u>>></u>
#5	TIME LIMIT EXCEEDED		<u>>></u>
#6	TIME LIMIT EXCEEDED		<u>>></u>
#7	ACCEPTED	0.05 s	<u>>></u>
#8	ACCEPTED	0.00 s	<u>>></u>
#9	ACCEPTED	0.00 s	<u>>></u>
#10	ACCEPTED	0.00 s	<u>>></u>
#11	ACCEPTED	0.57 s	<u>>></u>
#12	TIME LIMIT EXCEEDED		<u>>></u>
#13	TIME LIMIT EXCEEDED		>>

Submission details

lask:	Nearest Smaller Values	
Sender:	ariloc	
Submission time:	2025-05-31 07:06:22 +0300	
Language:	C++ (C++20)	
Status:	READY	
Result:	TIME LIMIT EXCEEDED	

Test results

test	verdict	time	
#1	ACCEPTED	0.00 s	<u>>></u>
#2	ACCEPTED	0.00 s	<u>>></u>
#3	ACCEPTED	0.00 s	<u>>></u>
#4	ACCEPTED	0.04 s	<u>>></u>
#5	TIME LIMIT EXCEEDED		<u>>></u>
#6	TIME LIMIT EXCEEDED		<u>>></u>
#7	ACCEPTED	0.05 s	<u>>></u>
#8	ACCEPTED	0.00 s	<u>>></u>
#9	ACCEPTED	0.00 s	<u>>></u>
#10	ACCEPTED	0.00 s	<u>>></u>
#11	ACCEPTED	0.57 s	<u>>></u>
#12	TIME LIMIT EXCEEDED		<u>>></u>
#13	TIME LIMIT EXCEEDED		<u>>></u>

Nuevo veredicto adquirido: Time Limit Exceeded (abreviado normalmente como TLE), o en español, tiempo límite excedido. Como dice su nombre, nos indica que en al menos uno de los casos nuestra solución tardó más de 1s en ejecutarse para la entrada dada.

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló.

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló. Más aún, se pueden ver y descargar los casos para verlos e intentar resolver los problemas en nuestro código.

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló. Más aún, se pueden ver y descargar los casos para verlos e intentar resolver los problemas en nuestro código. NO hay que acostumbrarse a esto, en las competencias NO podemos ver los casos de prueba, y tampoco en muchos jueces:

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló. Más aún, se pueden ver y descargar los casos para verlos e intentar resolver los problemas en nuestro código. NO hay que acostumbrarse a esto, en las competencias NO podemos ver los casos de prueba, y tampoco en muchos jueces:

 omegaUp nunca deja ver los casos, y a veces tampoco qué subtareas funcionaron, a menos que se haya resuelto el problema antes con 100ptos.

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló. Más aún, se pueden ver y descargar los casos para verlos e intentar resolver los problemas en nuestro código. NO hay que acostumbrarse a esto, en las competencias NO podemos ver los casos de prueba, y tampoco en muchos jueces:

- omegaUp nunca deja ver los casos, y a veces tampoco qué subtareas funcionaron, a menos que se haya resuelto el problema antes con 100ptos.
- CodeForces deja ver los casos de prueba sólo en modo práctica (no en competencia), e incluso así están recortados, impidiendo ver la totalidad de casos grandes.

A diferencia de CodeForces donde sólo se ejecutan los casos en orden y se detiene la ejecución apenas falla en alguno, en CSES se ejecutan todos los casos sin importar si alguno "anterior" falló. Más aún, se pueden ver y descargar los casos para verlos e intentar resolver los problemas en nuestro código. NO hay que acostumbrarse a esto, en las competencias NO podemos ver los casos de prueba, y tampoco en muchos jueces:

- omegaUp nunca deja ver los casos, y a veces tampoco qué subtareas funcionaron, a menos que se haya resuelto el problema antes con 100ptos.
- CodeForces deja ver los casos de prueba sólo en modo práctica (no en competencia), e incluso así están recortados, impidiendo ver la totalidad de casos grandes.
- El **OIA Juez** nunca deja ver los casos, aunque sí las subtareas, sus puntajes correspondientes al total obtenido, e incluso a veces el veredicto obtenido en los casos de la subtarea.

Finalmente, vamos a introducir un último veredicto no menos importante a los ya vistos: el **Runtime Error** (abreviado **RE** o **RTE**), o en español, *error* de ejecución.

Finalmente, vamos a introducir un último veredicto no menos importante a los ya vistos: el **Runtime Error** (abreviado **RE** o **RTE**), o en español, *error* de ejecución.

¿Alguna vez les pasó que ejecutaron un programa en C++ y terminó abruptamente con Segmentation fault (core dumped)?

Finalmente, vamos a introducir un último veredicto no menos importante a los ya vistos: el **Runtime Error** (abreviado **RE** o **RTE**), o en español, *error* de ejecución.

¿Alguna vez les pasó que ejecutaron un programa en C++ y terminó abruptamente con Segmentation fault (core dumped)? Si nuestro programa "crashea" ejecutando algún caso en el juez, el veredicto mostrado será entonces *Runtime Error*.

Finalmente, vamos a introducir un último veredicto no menos importante a los ya vistos: el **Runtime Error** (abreviado **RE** o **RTE**), o en español, *error* de ejecución.

¿Alguna vez les pasó que ejecutaron un programa en C++ y terminó abruptamente con Segmentation fault (core dumped)? Si nuestro programa "crashea" ejecutando algún caso en el juez, el veredicto mostrado será entonces *Runtime Error*.

El origen de un error de este tipo es claramente por un problema en la implementación de la solución, un bug en la programación. El más común de estos errores es "acceder a una posición fuera de memoria", por ejemplo, tener un vector<int> con tamaño n y querer acceder/escribir en algún punto a arr[n] (inválido porque está indexado en 0), o a arr[-1].

RE en algunos jueces

A veces los runtime error no están marcados de forma clara en los jueces. Presentamos algunos ejemplos para que sepan identificarlos:

RE en algunos jueces

A veces los runtime error no están marcados de forma clara en los jueces. Presentamos algunos ejemplos para que sepan identificarlos:

omegaUp: "Error de ejecución"



 CodeForces: "Runtime error on test Problem Verdict Lang



 OIA Juez: "Execution killed (could be triggered by violating memory) limits)"



SPOJ: "runtime error (SIGSEGV)"



Tenemos entonces los siguientes veredictos principales posibles:

 Accepted: Nuestra solución resuelve todos los casos de prueba correctamente y en los límites estipulados.

- Accepted: Nuestra solución resuelve todos los casos de prueba correctamente y en los límites estipulados.
- TLE: Nuestra solución excede el límite de *tiempo* estipulado en algún caso de prueba.

- Accepted: Nuestra solución resuelve todos los casos de prueba correctamente y en los límites estipulados.
- TLE: Nuestra solución excede el límite de tiempo estipulado en algún caso de prueba.
- MLE: Nuestra solución excede el límite de memoria estipulado en algún caso de prueba.

- Accepted: Nuestra solución resuelve todos los casos de prueba correctamente y en los límites estipulados.
- TLE: Nuestra solución excede el límite de tiempo estipulado en algún caso de prueba.
- MLE: Nuestra solución excede el límite de memoria estipulado en algún caso de prueba.
- WA: Nuestra solución no excede ninguno de los límites estipulados, pero escribe una respuesta incorrecta en alguno de los casos de prueba.

Tenemos entonces los siguientes veredictos principales posibles:

- Accepted: Nuestra solución resuelve todos los casos de prueba correctamente y en los límites estipulados.
- TLE: Nuestra solución excede el límite de tiempo estipulado en algún caso de prueba.
- MLE: Nuestra solución excede el límite de memoria estipulado en algún caso de prueba.
- WA: Nuestra solución no excede ninguno de los límites estipulados, pero escribe una respuesta incorrecta en alguno de los casos de prueba.

Y además, en problemas tipo IOI, los problemas pueden tener puntaje parcial, donde cada subtarea tiene su propio veredicto, y las subtareas AC otorgan puntos según sea indicado. El veredicto general que se muestra depende del juez, pero no tomará AC si no todas las subtareas son AC.

OJO!

OJO!

En algunos casos también las subtareas pueden tener puntaje parcial. Por ejemplo, el enunciado puede consistir en imprimir dos líneas, una con un resultado y otra con "una forma de obtener ese resultado" (según el contexto del problema), y se indica algo como que "Se otorga el 50 % del puntaje si se imprime correctamente la primer línea, y el 50 % restante si se imprime correctamente la segunda".

OJO!

En algunos casos también las subtareas pueden tener puntaje parcial. Por ejemplo, el enunciado puede consistir en imprimir dos líneas, una con un resultado y otra con "una forma de obtener ese resultado" (según el contexto del problema), y se indica algo como que "Se otorga el 50 % del puntaje si se imprime correctamente la primer línea, y el 50 % restante si se imprime correctamente la segunda".

En estos casos hay que prestar atención al enunciado, pero indica que también puede haber puntaje parcial más allá de las subtareas. Es decir, si el enunciado tiene un mensaje como el ejemplo anterior, suele pasar que el puntaje de cada subtarea se divide a la mitad, donde se otorga cada mitad del puntaje de la subtarea según las líneas que se hayan impreso correctamente

Por supuesto, hay que ver cada caso y leer el reglamento de la competencia (por ejemplo OIA).

- Introducción
- 2 Complejidad computaciona
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

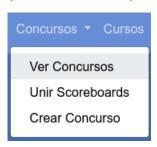
Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente.

Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.

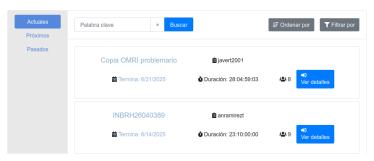
Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.

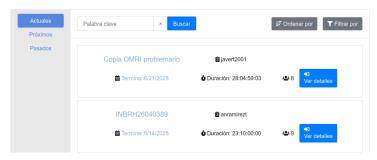


Previamente mencionamos que omegaUp permite crear competencias (llamados *concursos*) fácilmente. Probemos buscar un concurso.

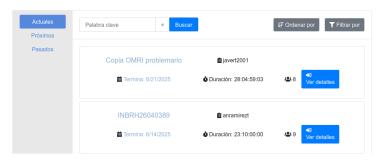


Al hacer click en la pestaña Concursos desde la página principal, podemos ver la lista de todos los concursos disponibles entrando a "Ver Concursos".



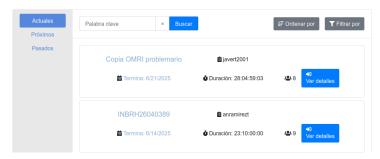


Al ingresar, nos encontraremos en una pantalla como esta. Aquí se encuentran las competencias en curso, ordenadas por tiempo de finalización de "más tarde a más temprano". Hay que considerar que una competencia se puede armar para que dure días o incluso meses.



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.



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).

¡Entremos a un concurso y mostremos cómo functiona!

- Introducciór
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

Hay para elegir

Como habrán visto, hay muchísimos jueces para elegir. Además de los ya presentados, recomendamos también los siguientes, donde algunos ya los mencionamos:

Hay para elegir

Como habrán visto, hay muchísimos jueces para elegir. Además de los ya presentados, recomendamos también los siguientes, donde algunos ya los mencionamos:

 OIA Juez: Contiene problemas tipo IOI de las distintas instancias de la Olimpíada Informática Argentina de distintos años (Prog. Competitiva nivel secundario). Tiene muchos problemas muy buenos, incluyendo algunos interesantes para arrancar. Personalmente recomendado.

Hay para elegir

Como habrán visto, hay muchísimos jueces para elegir. Además de los ya presentados, recomendamos también los siguientes, donde algunos ya los mencionamos:

- OIA Juez: Contiene problemas tipo IOI de las distintas instancias de la Olimpíada Informática Argentina de distintos años (Prog. Competitiva nivel secundario). Tiene muchos problemas muy buenos, incluyendo algunos interesantes para arrancar. Personalmente recomendado.
- CodeForces Gym: Además del set de problemas normal de CodeForces, la sección de "Gym" contiene, de forma similar a omegaUp, competencias subidas por los usuarios. Muchas de éstas incluyen copias de instancias regionales ICPC (Prog. Competitiva nivel universitario) de todo el mundo, incluyendo World Finals, siendo un recurso valiosísimo para entrenar en equipo.

• UVa Online Judge: Juez de antaño que data desde antes del año 2000. Contiene problemas de ICPC antiguos y un set de problemas diversos muy extenso. Muchos de sus problemas y soluciones están mencionados en la famosa serie de libros "Competitive Programming", de Steven y Felix Halim.

- UVa Online Judge: Juez de antaño que data desde antes del año 2000. Contiene problemas de ICPC antiguos y un set de problemas diversos muy extenso. Muchos de sus problemas y soluciones están mencionados en la famosa serie de libros "Competitive Programming", de Steven y Felix Halim.
- Kattis: Un juez más reciente con algunos problemas interesantes, también citado por Steven y Felix Halim en su edición más reciente de su serie de libros de programación competititya, el "Competitive" Programming 4".

Otros otros jueces

También mencionamos algunos jueces más con los cuales se pueden encontrar algún que otro problema de calidad, o problemas clásicos conocidos:

Otros otros jueces

También mencionamos algunos jueces más con los cuales se pueden encontrar algún que otro problema de calidad, o problemas clásicos conocidos:

- Sphere Online Judge (SPOJ): Conocido por problemas clásicos como DQUERY o KQUERY, al igual que algunos problemas viejos de instancias de ICPC de la UBA.
- HackerRank: Conocido por su facilidad para armar contests, pueden encontrárselo alguna vez para competencias con problemas personalizados.
- AtCoder: Juez de origen japonés con competencias regulares los sábados o domingos a la mañana (hora Argentina). Suele contener problemas distintos a otros lugares y algo creativos, con algunos problemas sencillos pero que suben de dificultad rápidamente.

- DMOJ: Conocido por contener problemas de algunas IOIs viejas.
- oj.uz: Conocido por contener problemas de distintas instancias regionales de competencias tipo IOI, incluyendo IOIs viejas.
- Meta Hacker Cup: Meta (Facebook) lleva a cabo una competencia de programación competitiva todos los años con problemas interesantes. Consultar el registro de problemas de años anteriores puede ser entretenido.

Comodín

También puede ser útil usar herramientas como vJudge, que recopilan problemas de múltiples jueces online, y permite crear competencias personalizadas con problemas de distintas fuentes.

- Introducción
- 2 Complejidad computacional
 - Un ejemplo
 - Definición
 - Chau constantes (o casi)
 - Algunos órdenes de complejidad comunes
 - Estimando el tiempo de ejecución
 - Análisis amortizado
- Jueces Online
 - Introducción
 - omegaUp: lo básico
 - Veredictos
 - omegaUp: competencias
 - Otros jueces
- 4 Cierre

¿Preguntas? ¿Dudas?

¡Muchas gracias!