Análisis de Algoritmos

Bach. Rodolfo Mercado Gonzales
Programación Competitiva UPC

Introducción

- Las computadoras pueden ser muy rápidas, pero no infinitamente rápidas.
- La memoria puede ser barata, pero no es gratuita.
- El tiempo de ejecución y el espacio de memoria son recursos limitados.

Análisis de algoritmos

- Busca estimar los recursos que un algoritmo requiere para poder ejecutarse.
- Nos permite comparar algoritmos y saber cuál es más eficiente.
- Nos centraremos en el tiempo de ejecución y el espacio de memoria.

Tiempo de ejecución

Tiempo de ejecución

Intervalo de tiempo que le toma a un programa procesar una determinada entrada.

Tiempo de ejecución

¿ Es un buen indicador medir el tiempo de ejecución en segundos, minutos, ... ?

✕

- Varía de acuerdo a la computadora que usemos para ejecutar el programa.
- Varía de acuerdo al tamaño de la entrada.
- Varía entre ejecución y ejecución.

El modelo RAM

- Para nuestro análisis definiremos una computadora teórica denominada Random Access Machine (RAM).
- Representa el comportamiento esencial de las computadoras.
- Las instrucciones son ejecutadas una después de otra, sin concurrencia.

El modelo RAM

Operaciones elementales

- Operaciones o instrucciones cuyo tiempo de ejecución no depende del tamaño de la entrada.
- Se realizan en tiempo unitario o constante.

los bucles y subprogramas poseen varias operaciones elementales.

- Operaciones aritméticas.
- Operaciones de comparación (datos primitivos).
- Asignar el valor a una variable (datos primitivos).
- Acceder a un elemento de un arreglo.
- Llamada a una función y retorno de un valor.

El modelo RAM

El tiempo de ejecución $T(n)$ de un programa se mide como el total de operaciones elementales realizadas para procesar una entrada de tamaño n.

Análisis del tiempo de ejecución

Hallemos la cantidad de operaciones elementales dentro de la función.

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

¿siempre hay n comparaciones?

Análisis del mejor caso

Menor número posible de operaciones elementales para una entrada de tamaño n.

comparación (1 op)

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

asignación (1 op)

retorno (1 op)

acceso (1 op)

comparación (1 op)

$$T(n) = 5$$

Análisis del peor caso

Máximo número posible de operaciones elementales para una entrada de tamaño n.

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

comparación (n + 1 op)

incremento (n op)

asignación (n op)

asignación (1 op)

retorno (1 op)

acceso (n op)

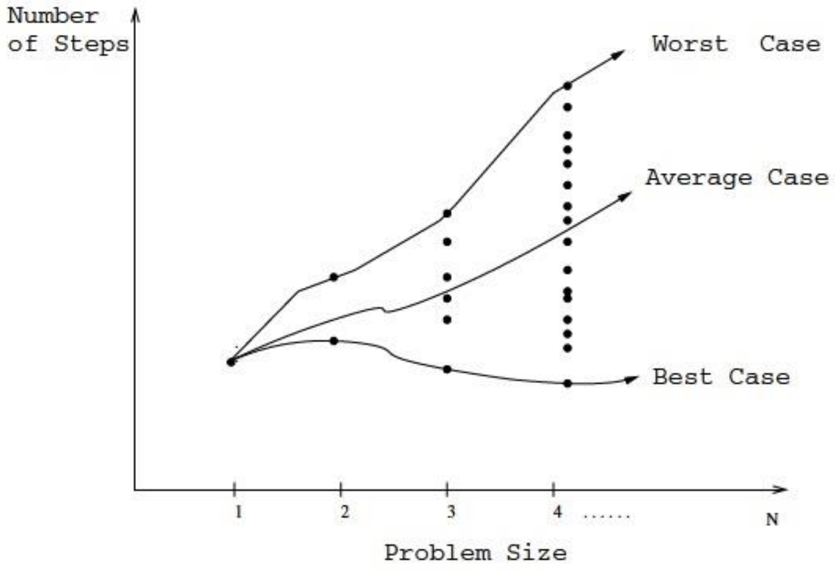
comparación (n op)

$$T(n) = 5n + 3$$

Análisis del caso promedio

- Número de pasos promedio para una entrada de tamaño n.
- Es complicado calcularlo.
- No lo usaremos por ahora.

Análisis del tiempo de ejecución



debemos analizar el peor de lo casos.

Dificultades

Calcular el número exacto de operaciones elementales requiere que el algoritmo sea especificado detalladamente.

Notación Big O

- La notación Big O es usada frecuentemente para denotar el tiempo de ejecución y la memoria usada en un algoritmo.

- Nos permite compara algoritmos sin necesidad de codificarlos.

- Nos brinda un cota superior para nuestra función $\boldsymbol{T(n)}$.

- Nos permitirá ignorar detalles que no alteran el comportamiento del tiempo de ejecución $\boldsymbol{T(n)}$.

Notación Big O

Dadas las funciones $T(n)$ y $g(n)$, se dice que $T(n)$ es $O(g(n))$ si existen dos constantes positivas $\boldsymbol{a}$ y $\boldsymbol{c}$ tal que:

$$T(n) \leq \boldsymbol{c} * g(n) \quad \text{para todo } n \geq \boldsymbol{a}$$

Notación Big O

Ejemplo

$Sea\ T(n\ ) = 5n + 3 \qquad \rightarrow$

Podemos decir que $T(n)$ es $\mathrm{O}(n)$, ya que
$5n + 3 \leq \mathbf{6} * n\ ,\ \ n \geq \mathbf{1}$

Podemos decir que $T(n)$ es $\mathrm{O}(n^2)$
$5n + 3 \leq \mathbf{4} * n^2\ ,\ \ n \geq \mathbf{2}$

Reglas de la notación Big O

$T(n) = 5n + 3$

- Eliminemos los términos de menor orden de nuestra función $T(n)$ $\rightarrow\ 5n$
- Eliminemos los factores constantes de nuestra función $T(n)$ $\rightarrow\ n$
- Usemos la función más pequeña posible para $g(n)$ $\rightarrow\ n\ es\ O(n)\ ya\ no\ O(n^2)$

enfoquémonos en la parte esencial de nuestro algoritmo

Notación Big O

$T(n) = 2n^2 + 100n + 6 \quad \rightarrow \quad \boldsymbol{O(n^2)}$ cuadrático

$T(n) = 5 \quad \rightarrow \quad \boldsymbol{O(1)}$ constante

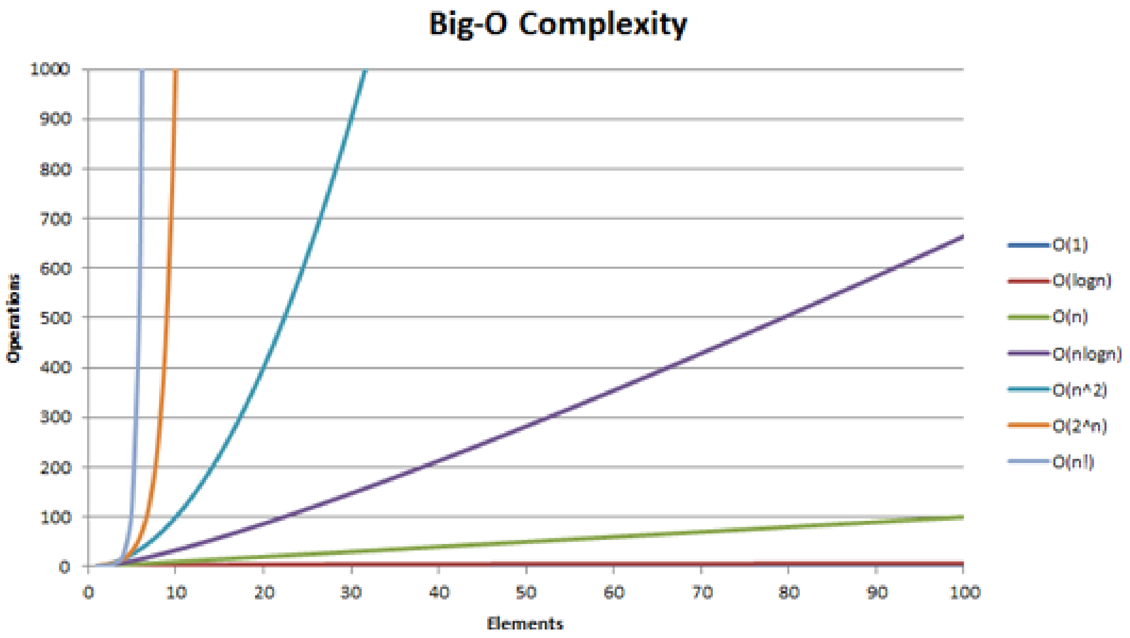
$T(n) = 3 * 2^n + 5 \quad \rightarrow \quad \boldsymbol{O(2^n)}$ exponencial

$T(n) = n + 6 \quad \rightarrow \quad \boldsymbol{O(n)}$ lineal

$T(n) = 2\log n + 1 \quad \rightarrow \quad \boldsymbol{O(log\ n)}$ logarítmico

$T(n, m) = 2 * n^2 + 3 * m \quad \rightarrow \quad \boldsymbol{O(n^2 + m)}$

Notación Big O



Consideraciones

- El tiempo de ejecución de un bucle se aproxima al número de veces (iteraciones) que el código dentro del bucle se ejecuta.

```
for (int i = 1; i <= n; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

$complejidad: O(n^2)$

Consideraciones

```
for (int i = 1; i <= n+5; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        ...
    }
}
```

$complejidad: O(n * m)$

```
for (int i = 1; i <= 3*n; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i *= 2) {
    ...
}
```

$complejidad: O(\log n)$

Consideraciones

```
for (int i = 1; i <= n; i++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

$complejidad: O(n^2)$

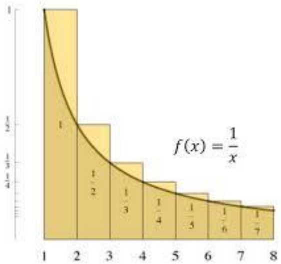
Consideraciones

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        ...
    }
}
```

$$T(n) \sim 1 + 2 + 3 + \cdots + n = \frac{1}{2}n^2 + \frac{1}{2}n$$

$$complejidad:\ O(n^2)$$

Consideraciones



$$\sum_{x=1}^{n} 1/x \sim \int_{1}^{n} 1/x \; dx = \log n$$

```
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j += i) {
        ...
    }
}
```

$$T(n) \sim n + \frac{n}{2} + \frac{n}{3} + \cdots + 1 = n * (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n})$$

$complejidad\text{: } O(n \log n)$

Consideraciones

- Debemos tener cuidado al trabajar con cadenas, las operaciones de asignación, comparación y concatenación son lineales en sus tamaños.

- No todas las funciones que tenemos disponibles en C++ son $O(1)$ como aparentan.

Función	Complejidad
sort	$O(n \log \mathrm{n})$
reverse	$O(n)$
insert (strings)	$O(n)$
replace (strings)	$O(n)$
$size$	$O(1)$
pop_back	$O(1)$
$back$	$O(1)$

Análisis del tiempo de ejecución

En el lenguaje **C++,** aproximadamente $\mathbf{10^8}$ operaciones elementales se ejecutan en 1 segundo.

```
clock_t ini = clock();
/*
    code
*/
clock_t fin = clock();
double run_time = (double) (fin - ini) / CLOCKS_PER_SEC;
cout << "runtime: " << fixed << setprecision(2) << run_time;
```

Análisis del tiempo de ejecución

Ejemplo 1

¿Cuántos múltiplos de 5 existen entre 1 y n $(n \leq 10^{10})$?

Análisis del tiempo de ejecución

Solución Ingenua

Recorremos cada uno de los números del 1 a n y revisamos si es divisible por 5.

complejidad: $O(n)$

Análisis del tiempo de ejecución

Solución Eficiente

$$multiplos\_cinco(1, n) = \lfloor n/5 \rfloor$$

complejidad: $O(1)$

Análisis del tiempo de ejecución

Ejemplo 2

Determinar si un número n ($n \leq 10^{10}$) es primo.

Análisis del tiempo de ejecución

Solución Ingenua

Tomando como caso especial que el 1 no es primo, recorremos cada uno de los números del 2 a $n-1$ (posibles divisores), si encontramos que alguno es divisor de n entonces el número no es primo, caso contrario será primo.

complejidad: $O(n)$

Análisis del tiempo de ejecución

Solución Eficiente

Teorema

Si n es un número compuesto, entonces n tiene al menos un divisor que es mayor que 1 y menor o igual a $\sqrt{n}$.

Análisis del tiempo de ejecución

Demostración

Sea $n = ab$; donde a, b son enteros, n un número compuesto y $1 < a \leq b < n$, entonces:

$\boldsymbol{a \leq \sqrt{n}}$, ya que si no caeríamos en una contradicción, porque tendríamos que $a, b > \sqrt{n}$ y por ende $ab > n$.

Asimismo

$\boldsymbol{b \geq \sqrt{n}}$, ya que si no caeríamos en una contradicción, porque tendríamos que $a, b < \sqrt{n}$ y por ende $ab < n$.

Análisis del tiempo de ejecución

Solución Eficiente

Por ende, para saber si un número $n > 1$ es primo, sólo es necesario verificar que no tenga divisores en el rango $[2, \sqrt{n}\,]$.

complejidad: $O(\sqrt{n}\,)$

Análisis del tiempo de ejecución

Ahora podemos tener una idea del orden de complejidad que requiere la solución a un problema dado una entrada de tamaño n.

Entrada	**Posible solución**
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n), O(2^n\ n)$
$n \leq 50$	$O(n^4)$
$n \leq 200$	$O(n^3)$
$n \leq 1000$	$O(n^2)$, $O(n^2 \log n)$
$n \leq 10^6$	$O(n), O(n \log n)$
$n \geq 10^9$	$O(1), O(\log n), O(\sqrt{n})$

límites comunes en los concursos de programación.

Ejercicios

- Codechef – Chef Jumping
- Codechef – Magic Pairs
- Codeforces – Single Push
- HackerRank – Strange Counter

Desafíos

- Codechef – Chef and Subarray
- Codechef – Count Substrings
- Hackerrank – Summing the N series
- Codechef – Isolation Centers
- Codechef – Please like me
- Codeforces – Sort the Array
- UVA – Code Refactoring
- Codechef – A problem onSticks
- Codeforces – Special Offer! Super Price 999 Bourles!

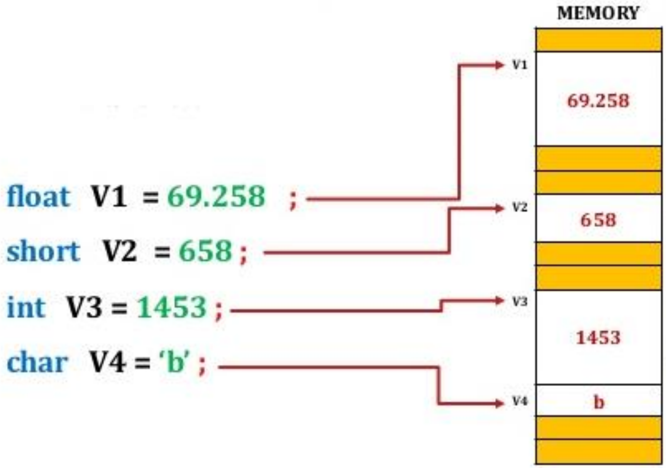
Desafíos

- [Codeforces – Sereja and Suffixes]
- [Codechef – Stone]
- [Codechef – Farmer Feb]
- [Codechef – Chef and Chain]
- [Codechef – Chef and the stones]
- [Codechef – Devu and an Array]
- [Codechef – Fun with Rotation]
- [Codechef – The Leaking Robot]
- [Codeforces – Mahmoud and a Triangle]

Espacio de memoria

¿Qué es una variable?

- Espacio en la memoria de la computadora donde se almacenará un valor.
- Posee un nombre (identificador) asociado.



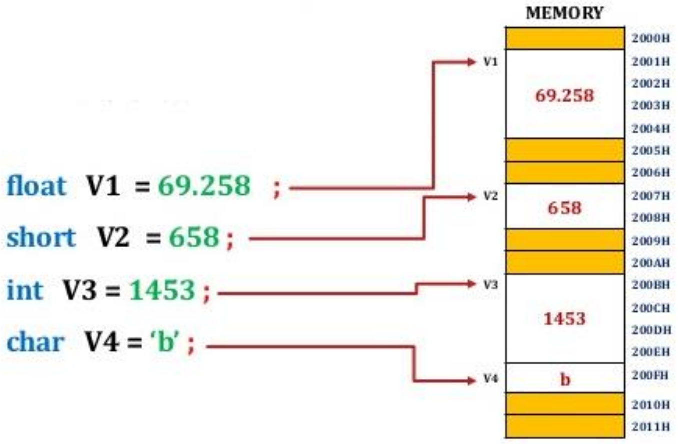
Memoria

- La memoria (RAM) es como un arreglo muy grande de celdas (bytes).

- A los índices (número hexadecimal) de la memoria, lo llamamos dirección de memoria.

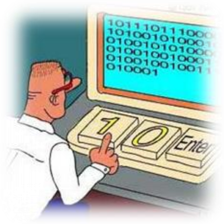
- Las variables ocupan una o más celdas de acuerdo a su tipo de dato.

direcciones

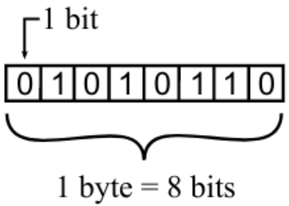


Bit

- Unidad mínima de información.
- La computadora “entiende” solo a nivel de bits.
- Es un dígito en el sistema binario, puede tomar los valores 0 (apagado) o 1 (prendido).



Byte (B)



- 1 byte (B) es una secuencia de 8 bits.
- Unidad usada para medir capacidad de almacenamiento de una memoria.

Binary			**Decimal**		
Name	**Symbol**	**Value (base 2)**	**Name**	**Symbol**	**Value (base 10)**
kibibyte	KiB	2^{10}	kilobyte	KB	10^{3}
mebibyte	MiB	2^{20}	megabyte	MB	10^{6}
gibibyte	GiB	2^{30}	gigabyte	GB	10^{9}
tebibyte	TiB	2^{40}	terabyte	TB	10^{12}
pebibyte	PiB	2^{50}	petabyte	PB	10^{15}
exbibyte	EiB	2^{60}	exabyte	EB	10^{18}

Representación de los números en un computador

- Los números son representados como una secuencia de bits.
- La cantidad de bits que se usan en su representación depende del tipo de dato.

tipo de dato	**bits / bytes en C++**
int, unsigned int, float	32 bits / 4 B
long long, unsigned long long, double	64 bits / 8 B

Enteros sin signo

- Permite almacenar solo números enteros no negativos (unsigned).
- Guarda el número en base binaria, completando con ceros a la izquierda.

decimal	unsigned int (32 bits)	unsigned long long (64 bits)
5	000……….00101	000000000……….00101
20	000……….10100	000000000……….10100

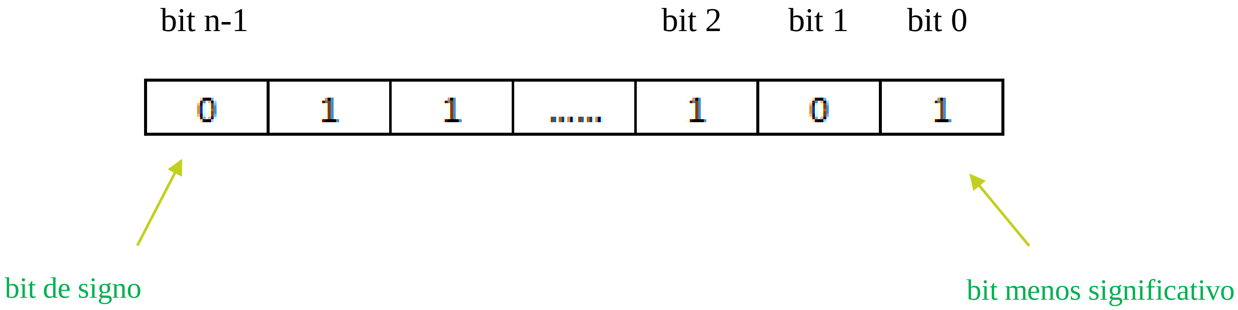
Enteros sin signo

- Con n bits podemos representar los enteros en el rango $[0, 2^n - 1]$.

unsigned int (32 bits)	unsigned long long (64 bits)
$[0, 2^{32} - 1]$	$[0, 2^{64} - 1]$

Enteros con signo

- ❑ Permite almacenar un número entero.
- ❑ Utiliza la representación complemento a dos.



Enteros con signo

Entero positivo o cero

El bit de signo es igual a 0 y los $n - 1$ bits restantes se completan con la representación binaria del número.

decimal	int (32 bits)	long long (64 bits)
0	000……….00000	000000000……….00000
20	000……….10100	000000000……….10100

Enteros con signo

Entero negativo

El bit de signo es igual a 1 y los $n-1$ bits restantes se completan con la representación binaria de $2^{n-1} - abs(x)$.

decimal	int (32 bits)	long long (64 bits)
-1	**1**11.........11111	**1**11111111.........11111
-3	**1**11.........11101	**1**11111111.........11101

Enteros con signo

- Con n bits podemos representar los enteros en el rango $[-2^{n-1}, 2^{n-1} - 1]$.

int (32 bits)	long long (64 bits)
$[-2^{31}, 2^{31} - 1]$	$[-2^{63}, 2^{63} - 1]$

Observaciones

- La representación de un entero negativo x en complemento a dos también se puede obtener con los siguientes pasos:

1. Hallamos la representación binaria de $abs(x)$ usando n bits.
2. Negamos todos los bits del número obtenido.
3. Finalmente le sumamos 1 al número obtenido.

Real con precisión simple

- Representado generalmente por el estándar IEEE 754.
- Se representan usando 32 bits (1 bit para indicar el signo, 8 bits para el exponente y 23 bits para la mantisa)

exponente

bits de signo

$$v = (-1)^{b_{31}} \times \underbrace{(1, b_{22} b_{21} \dots b_0)_2}_{\text{bits de mantisa}} \times 2^{e-127}$$

e varía entre 0 y 255

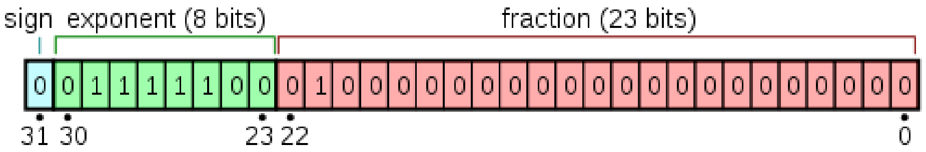
Tipo	Mínimo	máximo	precisión
float	$-3.4 * 10^{-38}$	$3.4 * 10^{38}$	7 cifras

Real con precisión simple

Ejemplo

$$\mathbf{0.15625} = 1.01 \, x \, 2^{-3}$$

signo: 0
mantisa: 0100….0
exponente: 124



Real con precisión doble

- Representado generalmente por el estándar IEEE 754.
- Se representan usando 64 bits (1 bit para indicar el signo, 11 bits para el exponente y 52 bits para la mantisa).
- Nos brinda el doble de precisión.

tipo	mínimo	máximo	precisión
double	$-1.7 * 10^{-308}$	$1.7 * 10^{308}$	15 cifras

Real con precisión doble

```
float f1 = 1.0/9;
float s1 = 0;
for (int i = 0; i < 10000; ++i)
    s1 += f1;
```

```
double f2 = 1.0/9;
double s2 = 0;
for (int i = 0; i < 10000; ++i)
    s2 += f2;
```

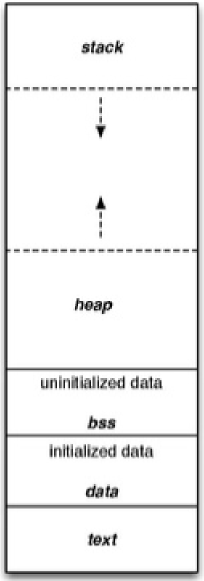
```
cout << fixed << setprecision(2) << s1 << " " << s2;
```

con double nuestro resultado es 1110.98, mientras que con float 1111.11.

Resumen tipo de datos

tipo de dato	bytes en C++	valores
bool	1 B	$0, 1$
char	1 B	$-128\ a\ 127$
int	4 B	-2147483648 a 2147483647
long long	8 B	$-9{,}223{,}372{,}036{,}854{,}775{,}808\ a$ $9{,}223{,}372{,}036{,}854{,}775{,}807$
float	4 B	$-3.4 * 10^{-38}\ \ a\ \ 3.4 * 10^{38}$
double	8 B	$-1.7 * 10^{-308}\ \ a\ 1.7 * 10^{308}$

Partes de la memoria



- **Text:** almacena el código en lenguaje máquina.
- **Data segment**: almacena las variables globales .
- **Stack**: almacena variables locales y llamadas a funciones.
- **Heap**

Data segment

Soporta más de los 256 MB permitidos en competencias.

```cpp
#include <bits/stdc++.h>
#define N 64000000
using namespace std;

int A[N];

int main() {
    A[N - 1] = 20;
    cout << A[N - 1];
    return 0;
}
```

Stack

- Región de la memoria que es gestionada eficientemente por el CPU.
- No necesitamos reservar ni liberar memoria manualmente.
- Almacena variables locales, parámetros y llamadas a funciones
- Tiene un tamaño limitado.

Stack

Por defecto tiene un límite de 2 MB.

```cpp
#include <bits/stdc++.h>
#define N 500000
using namespace std;

int main() {
    int A[N];
    A[N - 1] = 20;
    cout << A[N - 1];
    return 0;
}
```

podemos incrementar el tamaño del stack:

```
g++ -Wl,--stack=256000000 archivo.cpp
```

Heap

- Somos responsables de reservar y liberar memoria.
- La lectura y escritura es un poco más lenta que en el stack.
- No tiene límite de memoria.

Heap

Podemos usar toda la memoria que nos permita nuestro hardware.

```cpp
#include <bits/stdc++.h>
#define N 64000000
using namespace std;

int main() {
    int *arr = new int[N];
    arr[N - 1] = 20;
    cout << arr[N - 1];
    // delete[] arr;
}
```

Análisis del espacio de memoria

El espacio de memoria $S(n)$ de un programa se mide como la máxima cantidad de bytes que usamos, en un mismo instante, para procesar una entrada de tamaño n.

Análisis del espacio de memoria

Ejemplo

Si nuestro programa reserva 100 MB de memoria, luego libera unos 30 MB del espacio y finalmente vuelve a reservar 20 MB, entonces la memoria usada sería 100 MB.

Análisis del espacio de memoria

4 * n bytes

```
int A[n];
for (int i = 0; i < n; ++i) {
    cin >> A[i];
}
```

4 bytes

$$S(n) = 4n + 4$$

Notación Big O

También podemos usar la notación Big O para denotar el espacio de memoria de un programa.

4 * n bytes

```
int A[n];
for (int i = 0; i < n; ++i) {
    cin >> A[i];
}
```

4 bytes

$$S(n) = 4n + 4 \quad \rightarrow \quad \boldsymbol{O(n)}$$

Complejidad de un algoritmo

```
int A[ N ];

for( int i = 0; i < N; ++i ){
    for( int j = i + 1; j < N; ++j ){
        if( A[ i ] > A[ j ] ) swap( A[ i ], A[ j ] );
    }
}
```

tiempo: $O(N^2)$

espacio: $O(N)$

Referencias

- Thomas Cormen et al. - Introduction to Algorithms
- Steven Skiena - The Algorithm Design Manual
- Hackerearth - Memory Layout of C Program

“The only thing worse than starting something and failing is not starting something.”

- Seth Godin