# Complejidad

## Big O notation

Tomás Peiretti

# Complejidad temporal

La complejidad temporal permite estimar cuánto tiempo utilizará un algoritmo para una determinada entrada. La idea es representar la eficiencia como una función cuyo parámetro sea el tamaño de la entrada.

De esta manera, al calcular la complejidad temporal es posible saber si un algorimo es lo suficientemente rápido sin tener que implementarlo.

La complejidad temporal se denota $O(...)$ donde los tres puntos representan alguna función.

# Complejidad temporal: orden de magnitud

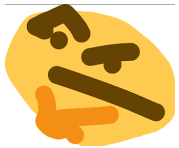
La complejidad temporal no nos dice cual es el numero exacto de veces que nuestro código se ejecutará, solo representa el orden de magnitud.

Por ejemplo, en los siguientes bucles el código se ejecuta **3n**, **n+5** y **n/2** veces respectivamente, pero la complejidad de cada uno sigue siendo O(n).

```
1  for (int i=0; i<3*n; i++) {
2      // .....
3  }
4
5  for (int i=0; i<n+5; i++) {
6      // .....
7  }
8
9  for (int i=0; i<n; i+=2) {
10     // .....
11 }
```

# Complejidad temporal: orden de magnitud

¿Qué complejidad tiene la siguiente función?



```
1 int sumaTriangular(int m[][1000], int n) {
2     int sum=0;
3     for(int i=0; i<n; i++) {
4         for (int j=i; j<n; j++) {
5             sum+=m[i][j];
6         }
7     }
8     return sum;
9 }
```

# Complejidad temporal: orden de magnitud

¿Qué complejidad tiene la siguiente función? $O(n^2)$

```
1 int sumaTriangular(int m[][1000], int n) {
2     int sum=0;
3     for(int i=0; i<n; i++) {
4         for (int j=i; j<n; j++) {
5             sum+=m[i][j];
6         }
7     }
8     return sum;
9 }
```

# Complejidad temporal: fases

Cuando un algoritmo consiste de múltiples fases (partes, funciones, etc), su complejidad será la complejidad de la fase que que requiere más tiempo.
Esto se debe a que la fase que requiere más tiempo (la de mayor complejidad) termina siendo el "cuello de botella" del algoritmo.

Por ejemplo, la siguiente función consiste de tres fases cuya complejidad es $O(n + n^2 + 1)$, por lo que su complejidad resultará de $O(n^2)$.

```
1  void miFunc(int & n) {
2      for (int i=0; i<n; i++) {
3          // .....
4      }
5
6      for (int i=0; i<n; i++) {
7          for (int j=0; j<n; j++)
8              // .....
9      }
10
11     n = -100;
12 }
```

# Complejidad temporal: múltiples variables

A veces, la complejidad del algoritmo depende de múltiples factores. Para estos casos, la función tendrá múltiples variables.

Por ejemplo, si debemos recorrer una matriz de N filas y M columnas la complejidad será O(N*M)

```
1 void imprimeMatriz(int mat[][1000], int n, int m) {
2     for (int i=0; i<n; i++) {
3         for (int j=0; j<n; j++)
4             cout << mat[i][j] << " ";
5         cout << endl;
6     }
7 }
```

# Complejidad temporal

¿Qué complejidad tiene el siguiente programa?

```
1  int sumaFila(int m[][1000], int fila, int tl) {
2      int sum=0;
3      for(int i=0; i<tl; i++) {
4          sum+=m[fila][i];
5      }
6      return sum;
7  }
8
9  void imprimeMediaMatriz(int mat[][1000], int n, int m) {
10     for (int i=0; i<n; i++)  {
11         for (int j=0; j<m/2; j++)
12             cout << mat[i][j] << " ";
13         cout << endl;
14     }
15 }
16
17 int main() {
18
19     int mat[2000][1000];
20     int n=100, m=50;
21
22     imprimeMediaMatriz(mat, n, m);
23     cout << sumaFila(mat, 0, m) + sumaFila(mat, n-1, m) << endl;
24 }
```

# Complejidad temporal

¿Qué complejidad tiene el siguiente programa? O(N*M)

```
1  int sumaFila(int m[][1000], int fila, int tl) {
2      int sum=0;
3      for(int i=0; i<tl; i++) {
4          sum+=m[fila][i];
5      }
6      return sum;
7  }
8
9  void imprimeMediaMatriz(int mat[][1000], int n, int m) {
10     for (int i=0; i<n; i++)  {
11         for (int j=0; j<m/2; j++)
12             cout << mat[i][j] << " ";
13         cout << endl;
14     }
15 }
16
17 int main() {
18
19     int mat[2000][1000];
20     int n=100, m=50;
21
22     imprimeMediaMatriz(mat, n, m);
23     cout << sumaFila(mat, 0, m) + sumaFila(mat, n-1, m) << endl;
24 }
```

# Complejidad temporal de funciones recursivas

La complejidad de una función recursiva se calcula como el producto entre el número de veces que la función se llama a sí misma y de la complejidad de una única llamada.

Por ejemplo, para la función que obtiene en n-ésimo término de la sucesión de fibonacci:

```
1 int fibonacci(int n) {
2     if (n == 1 || n == 2)
3         return 1;
4     return fibonacci(n-1) + fibonacci(n-2);
5 }
```

La llamada fibonacci(n) producirá $2^n$ llamadas y la complejidad de cada llamada será $O(1)$, por lo que la complejidad resultante será de $O(2^n)$

# Complejidad temporal de funciones recursivas

¿Qué complejidad tiene la siguiente función recursiva?

```
1  void imprimirALoLoco(int arr[], int n) {
2
3      if (n < 0) return;
4
5      for (int i=0; i<n; i++) {
6          cout << arr[i] << " ";
7      }
8      cout << endl;
9
10     imprimirALoLoco(arr, n-1);
11 }
```

# Complejidad temporal de funciones recursivas

¿Qué complejidad tiene la siguiente funcion? $O(n^2)$

```
1  void imprimirALoLoco(int arr[], int n) {
2
3      if (n < 0) return;
4
5      for (int i=0; i<n; i++) {
6          cout << arr[i] << " ";
7      }
8      cout << endl;
9
10     imprimirALoLoco(arr, n-1);
11 }
```

# Complejidades más usuales

$O(1)$ The running time of a **constant-time** algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer.

$O(\log n)$ A **logarithmic** algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because $\log_2 n$ equals the number of times $n$ must be divided by 2 to get 1.

$O(\sqrt{n})$ A **square root algorithm** is slower than $O(\log n)$ but faster than $O(n)$. A special property of square roots is that $\sqrt{n} = n/\sqrt{n}$, so the square root $\sqrt{n}$ lies, in some sense, in the middle of the input.

$O(n)$ A **linear** algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer.

$O(n \log n)$ This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is $O(n \log n)$. Another possibility is that the algorithm uses a data structure where each operation takes $O(\log n)$ time.

https://github.com/pllk/cphb/blob/master/book.pdf

# Complejidades más usuales

$O(n^2)$ A **quadratic** algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in $O(n^2)$ time.

$O(n^3)$ A **cubic** algorithm often contains three nested loops. It is possible to go through all triplets of the input elements in $O(n^3)$ time.

$O(2^n)$ This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of $\{1,2,3\}$ are $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ and $\{1,2,3\}$.

$O(n!)$ This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of $\{1,2,3\}$ are $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ and $(3,2,1)$.

`https://github.com/pllk/cphb/blob/master/book.pdf`