



Análisis de Algoritmos

complejidad en tiempo

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 usado.

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 definimos 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

❑ Se definen las operaciones elementales.

Operaciones o instrucciones cuyo tiempo de ejecución no depende del tamaño de la entrada:

- 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.



Los bucles y subprogramas poseen varias operaciones elementales.

El modelo RAM

- ❑ **El tiempo de ejecución se mide como la cantidad de operaciones elementales realizadas.**


Si asumimos que en el modelo RAM se ejecutan una cantidad de operaciones elementales por segundo, entonces ya podemos convertir el tiempo de ejecución según el modelo RAM en tiempo de ejecución real.



Análisis del tiempo de ejecución

Halleemos 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 tiempo de ejecución

Análisis del mejor caso

Menor 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;
}
```

Diagram illustrating the best-case time complexity analysis for the `search` function. The function is annotated with yellow arrows pointing to specific operations, each labeled with its cost in green text:

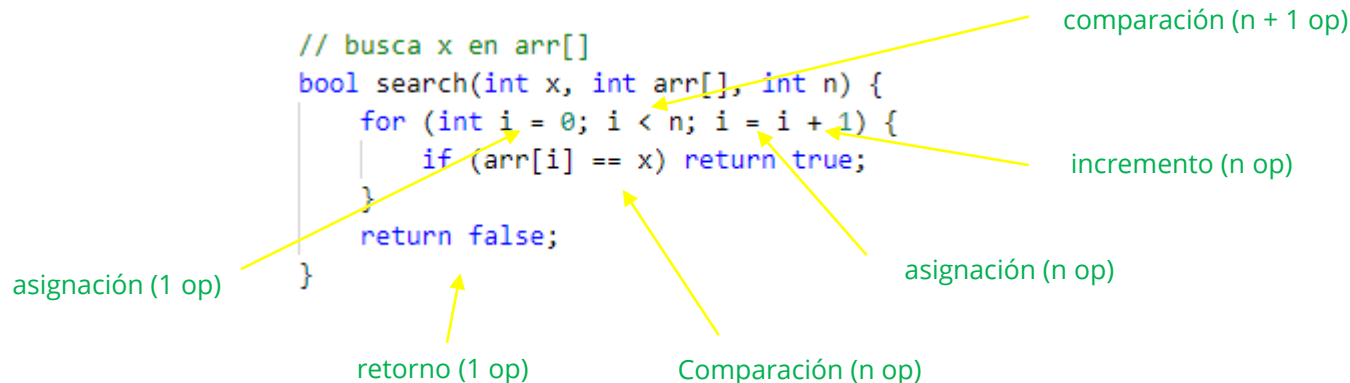
- asignación (1 op)**: Points to the assignment `i = 0` in the `for` loop header.
- comparación (1 op)**: Points to the comparison `i < n` in the `for` loop header.
- comparación (1 op)**: Points to the comparison `arr[i] == x` in the `if` statement.
- retorno (1 op)**: Points to the `return true` statement.
- Comparación (1 op)**: Points to the comparison `i = i + 1` in the `for` loop header.

$$T(n) = 4$$

Análisis del tiempo de ejecución

Análisis del peor caso

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



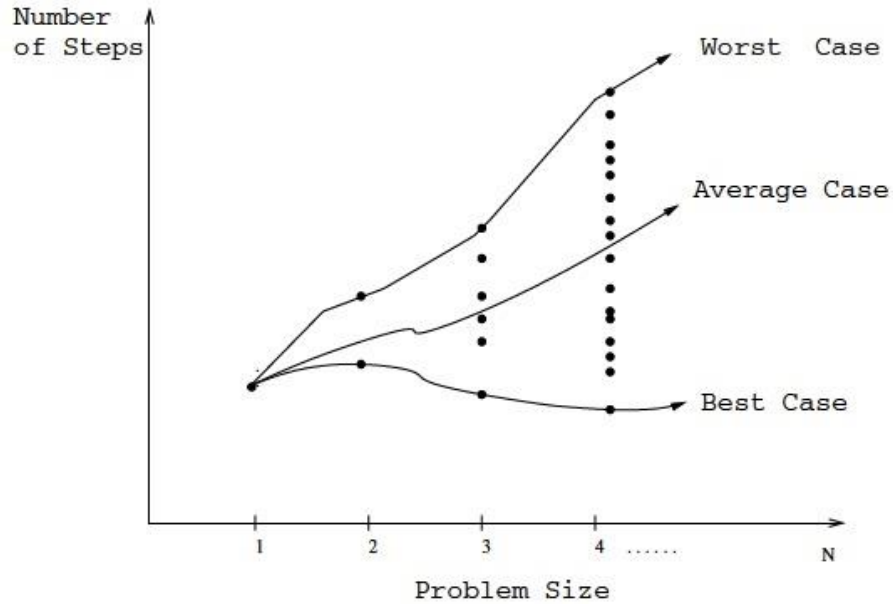
$$T(n) = 4n + 3$$

Análisis del tiempo de ejecución

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 los casos.



Dificultades

- ❑ Calcular el número exacto de operaciones elementales requiere que el algoritmo sea especificado detalladamente.
- ❑ Dos algoritmos pueden diferir en cantidad de operaciones elementales solo por detalles de implementación.

Notación Big O

- ❑ La notación **Big O** ignora detalles que no impactan en la comparación de algoritmos.
- ❑ Para hallar la complejidad $O(n)$ de un algoritmo sólo nos interesa el término de mayor orden de la función $T(n)$.



En competencias y en la industria denominamos big O a lo que a nivel científico se le llama big theta.

Notación Big O

$$T(n) = 2n^2 + 100n + 6 = \mathbf{O(n^2)}$$

$$T(n) = 5 = \mathbf{O(1)}$$

$$T(n) = 3 * 2^n + 5 = \mathbf{O(2^n)}$$

$$T(n) = n + 6 = \mathbf{O(n)}$$

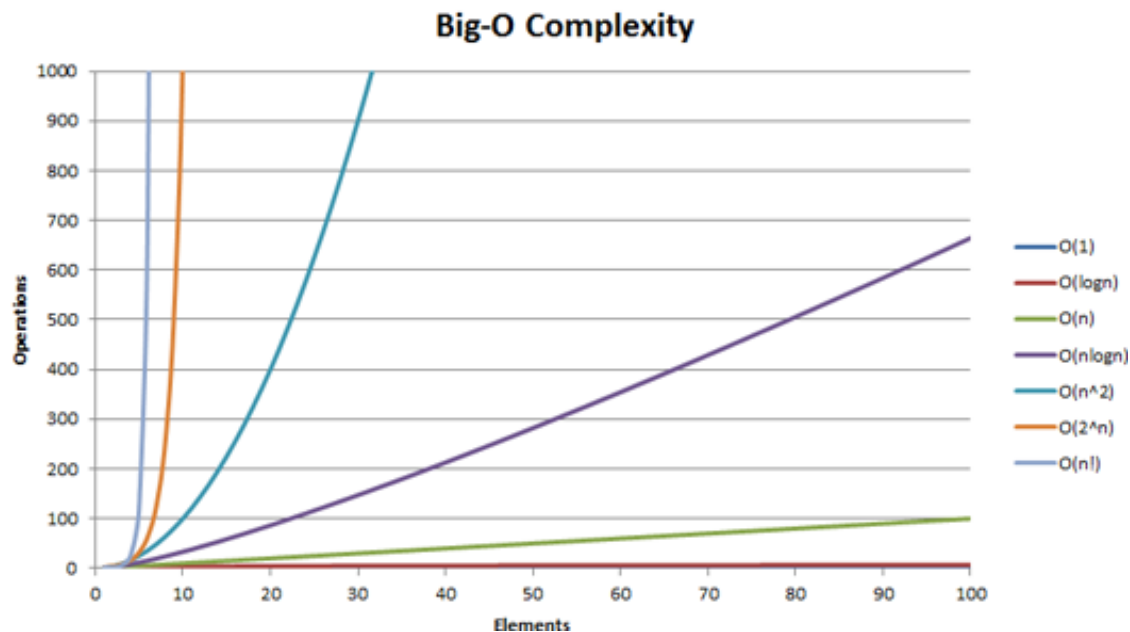
$$T(n) = 2 \log n + 1 = \mathbf{O(\log n)}$$

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

*Ahora ya podemos comparar
algoritmos por su “complejidad”*



Complejidad en tiempo



Análisis del tiempo de ejecución

En C++, aproximadamente 10^8 operaciones se ejecutan en 1 segundo.

```
clock_t ini = clock();
/*
   codigo
*/
clock_t fin = clock();
double tiempo = (double)(fin - ini) / CLOCKS_PER_SEC;
printf("tiempo : %.2f segundos", tiempo);
```

Complejidad en tiempo

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

Entrada	Posible solución
$n \leq 10$	$O(n!)$
$n \leq 22$	$O(2^n n)$
$n \leq 50$	$O(n^4)$
$n \leq 1000$	$O(n^2)$, $O(n^2 \log n)$
$n \leq 10^5$	$O(n)$, $O(n \log n)$
$n \leq 10^{18}$	$O(\log n)$, $O(1)$



Límites comunes en los concursos de programación.

Análisis de Algoritmos

Ejemplo 1

Hallar la suma de los múltiplos de 3 o 5 menores o iguales a n ($n \leq 10^9$).



Análisis de Algoritmos

Solución Ingenua

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

Complejidad en tiempo: **$O(n)$**

Análisis de Algoritmos

Solución Eficiente

$$\text{sum_multiples}(5) + \text{sum_multiples}(3) - \text{sum_multiples}(15)$$

$$\text{sum_multiples}(x) = x (1 + 2 + 3 + \dots + \lfloor n/x \rfloor)$$

Complejidad en tiempo: **$O(1)$**

Análisis de Algoritmos

Ejemplo 2

Determinar si un número n ($n \leq 10^9$) es primo.



Análisis de Algoritmos

Solución Ingenua

Tomando como un 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 en tiempo: **$O(n)$**

Análisis de Algoritmos

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 de Algoritmos

Solución Eficiente

Demostración

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

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

Análisis de Algoritmos

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 en tiempo: $O(\sqrt{n})$

Análisis de Algoritmos

Ejemplo 3

Ordenar de forma ascendente un arreglo de n ($n \leq 10^5$) números.



Análisis de Algoritmos

Solución Ingenua

Utilizamos el algoritmo “ordenamiento burbuja”.

Complejidad en tiempo: $O(n^2)$

Análisis de Algoritmos

Solución Eficiente

Probemos la función **sort** que se encuentra en el header **<algorithm>**.

Complejidad en tiempo: $O(n \log n)$

Problemas

[Codechef – Magic Pairs](#)

[Codechef – Chef Jumping](#)

[HackerRank – Strange Counter](#)

[Codechef – Reach The Point](#)

[Codeforces – Mahmoud and a Triangle](#)

Referencias

- ❑ Thomas Cormen et al. - **Introduction to Algorithms**
- ❑ Steven Skiena - **The Algorithm Design Manual**

¡ Good luck and have fun !