

Estructura de Datos y Algoritmos 1

Teórico #2:

Introducción al Análisis de Algoritmos (Tiempo de Ejecución)

Introducción

Para un problema pueden existir varios algoritmos distintos que lo solucionen.

Generalmente hemos elegido algoritmos sin preocuparnos cuán eficientes y cuán costosos son.

Es necesario conocer cómo analizar la **eficiencia** y el **costo en tiempo y en recursos** de un algoritmo determinado. De esa forma podemos comparar y escoger el algoritmo más adecuado para un determinado problema.

Criterios para analizar un algoritmo

1. Uso eficiente de los recursos.
2. Tamaño total del código: Cuánto ocupa en memoria.
3. Corrección: Si siempre produce resultados correctos para todos los posibles datos de entrada.
4. Legibilidad: Cuán fácil es de leer, entender y modificar.
5. Robustez: Cuán tolerante es a entradas erróneas o inesperadas.

El uso eficiente de los recursos se mide en:

1. **Complejidad espacial:** La memoria que el algoritmo utiliza durante su ejecución.
2. **Complejidad temporal:** El tiempo que tarda su ejecución.
(Criterio más usado para analizar algoritmos)

Tiempo de ejecución: Factores

1. **El algoritmo en sí:** Pueden existir dos algoritmos para un mismo problema de modo que uno sea más rápido que otro. Ej. Búsqueda Secuencial y Búsqueda Binaria para listas ordenadas.
2. **Los datos de entrada:** Para algunos datos de entrada el algoritmo puede demorarse más que con otros. Ejemplo: Algoritmo para calcular la potencia n -ésima de un número x .
3. **Computadora empleada para su ejecución:** Depende de la velocidad de su procesador, memoria disponible, etc.

Dos posibles análisis de la complejidad temporal

- 1. Análisis a posteriori:** Proporciona una medida real. Consiste en medir el tiempo de ejecución del algoritmo para una instancia de un problema y una computadora en concreto.
- 2. Análisis a priori:** Proporciona una medida teórica. Consiste en obtener una función que acote superior o inferiormente el tiempo de ejecución del algoritmo para cualquier instancia del problema.

Análisis a posteriori (medida real)

Se implementa el algoritmo en un lenguaje de programación y se mide el tiempo de ejecución usando funciones que devuelven el tiempo marcado por el reloj de la máquina:

```
#include <time.h>
int main()
{
    time_t tiempo_inicial, tiempo_final;
    tiempo_inicial = time(NULL);
    //Implementación del algoritmo
    tiempo_final = time(NULL);
    tiempo_ejecucion = tiempo_final - tiempo_inicial;
}
```

Desventaja: Es un método poco riguroso. Depende en gran medida de la computadora, el lenguaje de programación, datos de entrada.

Análisis a priori (medida teórica)

El tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales (básicas) que realiza el algoritmo para un tamaño de entrada dado.

La función del tiempo de ejecución se nombrará $T(n)$, donde n es el tamaño de los datos de entrada.

El algoritmo es ejecutado sobre un modelo de computadora MAD (Modelo de Acceso Directo)

Análisis a priori (medida teórica)

Máquina de Acceso Directo (MAD)

En ella se supone que:

1. El algoritmo es implementado utilizando su conjunto básico de instrucciones.
2. Las instrucciones son ejecutadas una a continuación de otra.
3. Para cada operación básica se conoce su tiempo de ejecución o su forma de calcular.

Tamaño de la entrada

Es el número de componentes sobre los que se va a ejecutar el algoritmo.

Ejemplo: La cantidad de elementos de una lista a ordenar, la dimensión de dos matrices a multiplicar.

Variantes de Análisis a priori

1. **Tiempo en el caso mejor:** Se corresponde con el menor de los tiempos requeridos por el algoritmo para ejecutarse con cada una de las instancias del problema.
2. **Tiempo en el caso peor:** Se corresponde con el mayor de los tiempos requeridos por el algoritmo para ejecutarse con cada una de las instancias del problema.
3. **Tiempo en el caso promedio:** Se corresponde con la división de la suma de los tiempos requeridos por el algoritmo para ejecutarse con cada una de las instancias del problema entre el total de las instancias.

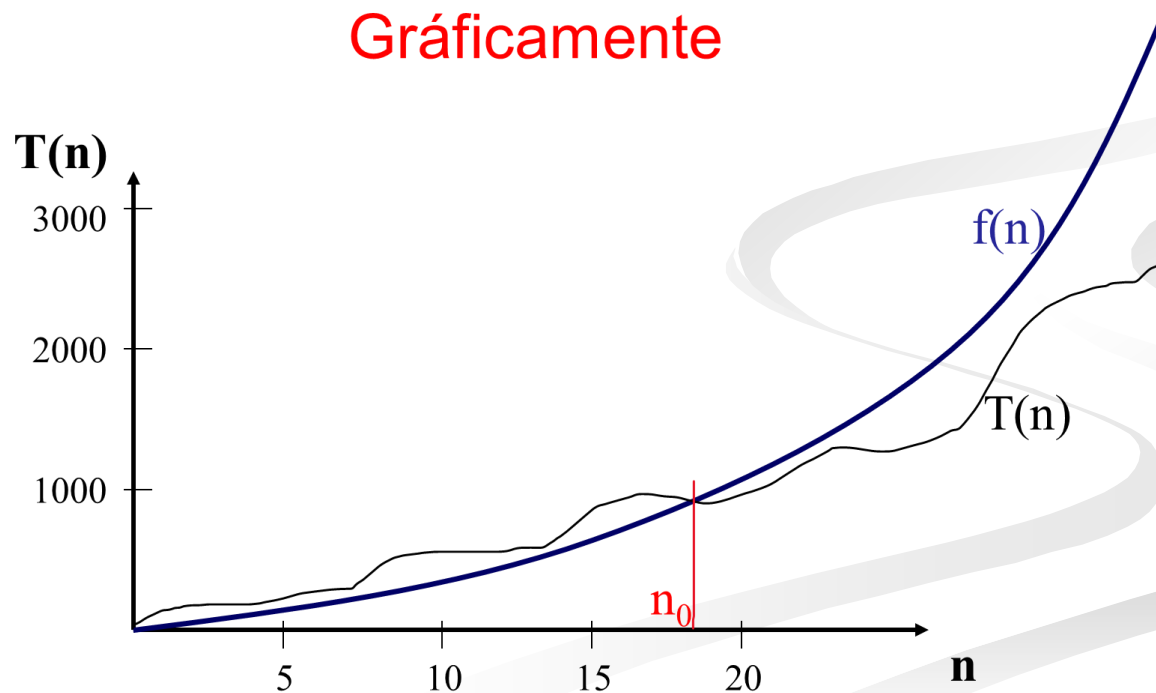
Cota Superior del Tiempo de Ejecución

Definición: $f(n)$ es cota superior de $T(n)$

($T(n)$ es «orden $f(n)$ ») si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$.

Notación: $T(n) = O(f(n))$

Gráficamente



Cota Superior del Tiempo de Ejecución

Definición: $f(n)$ es **cota superior** de $T(n)$

($T(n)$ es «orden $f(n)$ ») si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$.

Notación: $T(n) = O(f(n))$

Ejemplo 1: $T(n) = 3n + 25$, verifiquemos que $T(n) = O(n)$.

Debemos encontrar constantes positivas c y n_0 tal que para todo $n \geq n_0$ se cumple que $3n + 25 \leq cn$.

En efecto, si tomamos a $n_0 = 1$ y a $c = 28$, vemos que para todo $n \geq 1$ se cumple que $3n + 25 \leq 28n$

Cota Superior del Tiempo de Ejecución

Definición: $f(n)$ es **cota superior** de $T(n)$

($T(n)$ es «orden $f(n)$ ») si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$.

Notación: $T(n) = O(f(n))$

Ejemplo 2: $T(n) = 4n^2 + 3n + 5$,
verifiquemos que $T(n) = O(n^2)$.

Debemos encontrar constantes positivas c y n_0 tal que para todo $n \geq n_0$ se cumple que $4n^2 + 3n + 5 \leq cn^2$.

En efecto, si tomamos a $n_0 = 1$ y a $c = 12$, vemos que para todo $n \geq 1$ se cumple que $4n^2 + 3n + 5 \leq 12n^2$

Algunas propiedades de la Cota Superior

1) Si $f = O(g) \wedge f = O(h) \Rightarrow f = O(\min(g, h))$

2) Regla de la suma:

Si $f_1 = O(g) \wedge f_2 = O(h) \Rightarrow f_1 + f_2 = O(\max(g, h))$

3) Regla del producto:

Si $f_1 = O(g) \wedge f_2 = O(h) \Rightarrow f_1 \times f_2 = O(g \times h)$

Reglas para calcular complejidad temporal

Regla No. 1: Operaciones elementales

El tiempo requerido para:

- Acceder a un valor almacenado
- Almacenar un valor en memoria
- Realizar una operación aritmética (+, -, *, /, %)
- Realizar una operación lógica (&&, ||)
- Realizar una operación de comparación (<, >, ≤, ≥, =, ≠)
- Retornar un valor

Tiene complejidad $O(1)$

Reglas para calcular complejidad temporal

Regla No. 1: Operaciones elementales

Ejemplo 1: La instrucción:

$y = x;$

tiene complejidad $O(1)$

Ejemplo 2: La instrucción:

$y = x + 5;$

tiene complejidad $O(1)$

Reglas para calcular complejidad temporal

Regla No. 2: El tiempo requerido para la ejecución de un algoritmo de secuencia lineal es la sumatoria de los tiempos de cada instrucción.

Ejemplo: El algoritmo :

1. Leer(x);
2. cuadrado = x*x;
3. Mostrar(cuadrado);

tiene complejidad $O(1)$, pues

- Las instrucciones 1 y 3 tienen complejidad $O(1)$.
- La instrucción 2 tiene complejidad $O(1)$.

y $O(1) + O(1) + O(1) = O(1)$

Reglas para calcular complejidad temporal

Regla No. 3: El tiempo requerido para la ejecución de una instrucción condicional if C then I1 else I2, es la suma del tiempo necesario para evaluar la expresión C más el tiempo máximo de ejecución de los bloques asociados a las ramas then y else.

$$T = T(C) + \text{Máximo}(T(I1), T(I2))$$

Ejemplo: La instrucción :

```
if (opcion == 2) return x*x  
else return x*x*x;
```

Tiene complejidad $O(1)$, pues

- La expresión condicional tiene complejidad $O(1)$.
- La instrucción de la cláusula **else** tiene complejidad $O(1)$

Reglas para calcular complejidad temporal

Regla No. 4: El tiempo requerido para la ejecución de una instrucción condicional múltiple switch (E) es la suma del tiempo necesario para evaluar la expresión E, más el tiempo máximo de los bloques asociados a los valores, más el tiempo asociado a cada uno de las expresiones case.

Ejemplo: La instrucción :

```
switch(opcion)
{
    case 1 : return x*x;
    case 2 : return x*x*x;
    default : return x;
};
```

tiene complejidad $O(1)$

Reglas para calcular complejidad temporal

Regla No. 5: El tiempo requerido para la ejecución de una instrucción repetitiva while (C) do I, es la suma del tiempo necesario para la evaluación de la expresión C más el numero de iteraciones multiplicado por el tiempo para la Instrucción, sumado con el tiempo necesario para la evaluación de la expresión por la cantidad de iteraciones.

$$T = T(C) + (\text{iteraciones}) * (T(C) + T(I)).$$

Reglas para calcular complejidad temporal

Ejemplo: El algoritmo:

```
1. i = 0;  
2. while (i < n)  
3.   Mostrar(s[i++]);
```

tiene complejidad $O(n)$, pues

- La instrucción 1 tiene complejidad $O(1)$
- La condición tiene complejidad $O(1)$
- El ciclo tiene n iteraciones.
- La instrucción 3 tiene complejidad $O(1)$

$$O(1) + n(O(1) + O(1)) = O(1) + nO(1) = O(n)$$

Reglas para calcular complejidad temporal

Regla No.6: El tiempo requerido para la ejecución de una instrucción repetitiva for(I1, C, I2) I3 es equivalente al tiempo empleado en ejecutar la siguiente secuencia de instrucciones:

```
I1;  
while( C ) do {  
    I3;  
    I2;  
}
```

Regla No. 7: El tiempo requerido para la ejecución repetitiva do I while (C) es equivalente al tiempo empleado en ejecutar el bloque de instrucciones siguiente:

```
I;  
while( C ) do  
    I;
```

Reglas para calcular complejidad temporal

Regla No. 8: Si una determinada función \mathbf{P} tiene un tiempo de $O(f(n))$, con n a la medida del tamaño de los argumentos, cualquier función que llame a \mathbf{P} tiene en la llamada una cota de $O(f(n))$.

Evaluación de programas

Un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con $O(n^3)$ para resolver el mismo problema.

Supongamos dos programas P1 y P2 con $T_1(n) = 100n^2$ y $T_2(n) = 5n^3$

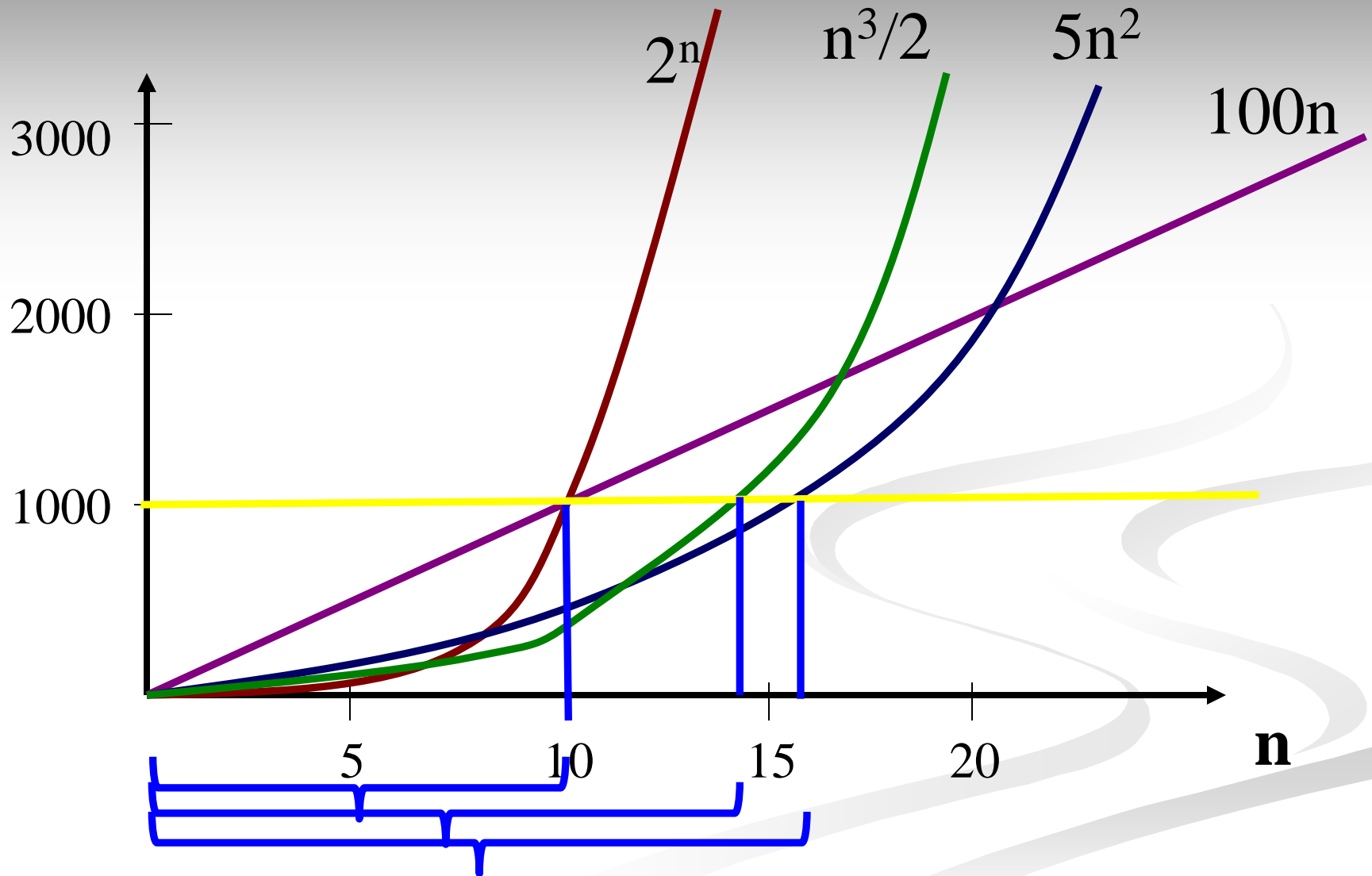
¿Cuál de los dos es preferible?

- Si $n < 20$ P2 es más rápido que P1
(para entradas “pequeñas” es mejor P2)
- Si $n > 20$ P1 es más rápido que P2
(para entradas “grandes” es mejor P1)

Evaluación de programas (cont.)


- La velocidad de crecimiento de un programa determina el tamaño de los problemas que se pueden resolver en un computador.
- Si bien las computadoras son cada vez más veloces, también aumentan los deseos de resolver problemas más grandes.
- Salvo que los programas tengan una velocidad de crecimiento baja, ej: $O(n)$ u $O(n \log n)$, un incremento en la rapidez del computador no influye significativamente en el tamaño de los problemas que pueden resolverse en una cantidad fija de tiempo.

Comparación de tiempos de ejecución



Cotas usuales en la complejidad temporal

Para n
grande



- $O(1)$: Constante
- $O(\log n)$: Logarítmico
- $O(\log^2 n)$: Log-cuadrado
- $O(n)$: Lineal
- $O(n \log n)$
- $O(n^2)$: Cuadrático
- $O(n^3)$: Cúbico
- $O(c^n)$: Exponencial

Ejemplos

```
... ..  
for (int i = 0; i < n; i++)  
    count += s[i];  
... ..  
for (int i = 0; i < n; i++)  
    if (i%2 == 0)  
        s[i] += count;  
... ..
```

$O(n)$

$O(n)$

$O(n)$

Ejemplos

... ..

```
int i = 1;
```

```
while (i<=n)
```

```
{
```

```
  for(int j=0; j<n; j++)
```

```
    for(int k=0; k<n; k++)
```

```
      s[j][k] *= s[j][k];
```

```
  s[i][i] -= s[i][i];
```

```
  i++;
```

```
}
```

$O(n)$ $nO(n)$ $nO(n^2)$

$$T(n) = O(n^3)$$

Ejemplos

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

} $O(\log n)$

Ejemplos

¿Qué orden de tiempo de ejecución tendrían los algoritmos para los siguientes problemas?

1. Calcular el máximo de dos números.
2. Calcular la sumatoria de los elementos de un array de tamaño n de números enteros.
3. Determinar el mayor elemento en un vector de tamaño n
4. Calcular la suma de los elementos de una matriz cuadrada $n \times n$

Ejercicio 1

Dadas las siguientes expresiones que indican el tiempo de ejecución de ciertos algoritmos, exprese la complejidad temporal en función de la cota superior “O grande”.

a) $T(n) = n^2 + n \cdot \log n$

b) $T(n) = n \cdot \log n + n + n \cdot \log n^2$

c) $T(n) = 3n + \sqrt[3]{8n^6} + n^2$

d) $T(n) = 1 + n(n+1) + 4^{\frac{n}{2}}$

Ejercicio 2:

Analizar tiempo de ejecución

```
bool esPrimo(int N){  
    int cont = 0;  
    for (int i = 2; i < N; i++)  
        if (N % i == 0)  
            return false;  
    return true;  
}
```


Ejercicio 3:

Analizar tiempo de ejecución

```
bool esPrimo(int N){  
    int cont = 0;  
    for (int i = 2; i < sqrt(N); i++)  
        if (N % i == 0)  
            return false;  
    return true;  
}
```

Ejercicio 4: Algoritmo para resolver un Sistema de Ecuaciones Lineales triangular superior

$$\begin{bmatrix} U_{11} & U_{12} & \dots & U_{1n} \\ & U_{22} & \dots & U_{2n} \\ & & \dots & \\ & & & U_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

```
void solveSELTs(float[][] U, float[] b, float[] x,
               int n){
    for (int i = n-1; i >= 0; i--) {
        x[i] = b[i] / U[i][i];
        for (int j = 0; j < i; j++)
            b[j] = b[j] - U[j][i]*x[i];
    }
}
```

Ejercicio 5: Multiplicación de matrices cuadradas

```
void dotMatrixMatrix(float[][] A, float[][] B,  
                    float[][] C, int n){  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++) {  
            C[i][j] = 0;  
            for (int k = 0; k < n; k++)  
                C[i][j] += A[i][k] * B[k][j];  
        }  
}
```

Ejercicio 6: Cantidad de cifras de un número natural

```
int cantidadCifras(int N){  
    int cont = 0;  
    while (N != 0){  
        N = N / 10;  
        cont++;  
    }  
    return cont;  
}
```