

Introducción a los Algoritmos y Complejidad Computacional

Aldo Sayeg Pasos Trejo

Algoritmos Computacionales
Facultad de Ciencias
Universidad Nacional Autónoma de México

12 de febrero de 2020

¿Cómo realizar un cómputo en la arquitectura de von Neumann?

- 1 Especificar el programa, es decir, las operaciones a realizar en el input en términos de las operaciones posibles en el ALU.
- 2 Especificar el input en la memoria.

Programa vs Algoritmo

Hemos usado como sinónimos las palabras **programa** y **algoritmo**. Matemáticamente, ambas son sinónimos y se refieren a un proceso computable que se puede realizar en un modelo de computabilidad (Máquina de Turing, Máquina de acceso aleatorio, etc)

En la práctica, existe una distinción sutil entre ambas

- **Algoritmo** se utiliza para referirse a un procedimiento computable para resolver un problema (por lo general matemático o computacional) específico.
- **Programa** se refiere más a la implementación en algún lenguaje de programación de un algoritmo o de cualquier otro proceso computable.

Programa vs Algoritmo

En este curso, nos enfocaremos tanto en algoritmos para resolver problemas de la física como en su realización en un programa.

Los algoritmos por lo general se tratan más bien como entes matemáticos abstractos que como realizaciones reales en un lenguaje. Sin embargo, su estudio por lo general está enfocado en resolver un problema aplicable a la realidad.

Errores en los programas

Cuando querramos hacer un programa que resuelva un problema (es decir, que dado un input genere el output deseado), es altamente probable que el programa **no funcione**. Las razones comunes son

- 1 (90 %) Errores de sintaxis u similares en el lenguaje de programación del programa.
- 2 (10 %) Errores en los algoritmos del programa.
 - 1 5 % No consideramos ciertos casos límites.
 - 2 5 % Problemas dependientes del algoritmo

Errores de sintaxis

Los errores de sintaxis son sumamente comunes y el acto de corregirlos (llamado coloquialmente **debugear**) toma la mayor parte de tiempo de escribir un programa.

Esos errores le suceden a todos los programadores y nos sucederán a nosotros de manera muy común. Son inevitables.

Errores en los algoritmos

Los algoritmos que veremos en la clase son sumamente sencillos, por lo que es particularmente poco probable encontrar errores en ellos

Dado un algoritmo, es posible demostrar que no contiene errores. Es decir, que dado un input con ciertas características, siempre obtendremos el output deseado. A esta propiedad se le llama **correctud**.

¿Realmente lo único importante de un algoritmo es su correctud?

En la teoría de algoritmos, no solo se pretende encontrar un algoritmo para cierto problema (por lo general difícil) y probar su correctud. Es fundamental también que los algoritmos sean realizables en términos de su **tiempo de ejecución** y sus **recursos de memoria**

Si un algoritmo resuelve un problema, pero tarda un tiempo muy grande en realizarse, entonces es posible que dicha solución no sea útil y práctica en la vida cotidiana.

A su vez, si para realizar un proceso en particular necesitamos que la computadora utilice una cantidad de memoria demasiado grande, es posible que se pueda realizar en ninguna computadora existente o sea demasiado caro.

¿Cómo medimos el tiempo y el espacio en memoria que necesita un algoritmo?

Formalmente, el tiempo y la memoria de un algoritmo depende del modelo de computabilidad que estemos usando. En la práctica y la teoría de algoritmos, se realiza una convención similar a la del pseudocódigo

- Asumimos que las operaciones básicas toman cierto tiempo, que medimos en unidades arbitrarias.
- Asumimos que cada tipo representable en la computadora toma una unidad de memoria dada.

Medición del tiempo y el espacio

Análogo a física, las unidades en las que medimos el tiempo y el espacio dependen del problema que estamos trabajando.

Por ejemplo, si estamos trabajando con un algoritmo que emplea enteros, podemos considerar que sumar dos enteros toma tiempo constante y que ese tiempo es una unidad arbitraria. También podemos considerar que almacenar un entero pesa una unidad de memoria.

Complejidad

A la medición del tiempo de ejecución y espacio en memoria de los algoritmos se le llama **complejidad temporal** (o solo “complejidad”) y **complejidad espacial**.

Dado un problema computacional definido con input y output deseado, las complejidades de un algoritmo que lo resuelva por lo general dependerán (i.e. serán función) de algunas características del input.

El análisis de la complejidad de un algoritmo corresponde en expresar, de manera aproximada pero explícita, dicha función.

Ejemplo de juguete: huevos a la mexicana

Algoritmo 1: Algoritmo para hacer huevos a la mexicana

Input : Jitomate, cebolla, chile serrano y huevos

Output: Huevo a la mexicana

- 1 Cortar el jitomate
- 2 Cortar la cebolla
- 3 Cortar el chile serrano
- 4 Saltear la cebolla en un sartén con aceite
- 5 Añadir el chile y el tomate
- 6 Añadir huevos crudos y batidos
- 7 **while** *No este cocinado el huevo* **do**
- 8 | Mover la mezcla en el sartén
- 9 **end**

Complejidades del algoritmo 1

Hipótesis

- Consideramos que tanto el tiempo como el espacio necesario para cortar jitomate, cebolla y chile es proporcional al volumen de cada uno e independiente entre sí.
- Mezclar, mover o añadir cosas toma el mismo tiempo independientemente del volumen.
- El tiempo y el espacio se miden en unidades proporcionales a los volúmenes.

Conclusiones

$$T(V_{\text{jitomate}}, V_{\text{cebolla}}, V_{\text{chile}}, N_{\text{huevos}}) \sim V_{\text{jitomate}} + V_{\text{cebolla}} + V_{\text{chile}} + N_{\text{huevos}}$$

$$E(V_{\text{jitomate}}, V_{\text{cebolla}}, V_{\text{chile}}, N_{\text{huevos}}) \sim E_{\text{jitomate}} + E_{\text{cebolla}} + E_{\text{chile}} + E_{\text{huevos}}$$

Ejemplo simple: sumar enteros

Algoritmo 2: Sumar lista de enteros

Input : Un arreglo de enteros A

Output: Un entero $suma$

```
1 Sea  $suma = 0$ 
2 Sea  $n = longitud(A)$ 
3 Sea  $i = 1$ 
4 while  $i \leq n$  do
5   |    $suma = suma + A[i]$ 
6   |    $i = i + 1$ 
7 end
```

Complejidades del algoritmo 2

Hipótesis

- Sumar dos enteros toma una cantidad constante de tiempo y es la unidad usada.
- Almacenar dos enteros toma una cantidad constante de tiempo y es la unidad usada

Conclusiones

$$T(A) \sim \text{longitud}(A) = n$$

$$E(A) \sim \text{longitud}(A) = n$$

Ejemplo real: calcular interacciones electrostáticas

Algoritmo 3: Cálculo de energía electrostática

Input : Un arreglo de vectores de \mathbb{R}^3 *Posiciones*, Un arreglo de flotantes \mathbb{R}^3 *Cargas*

Output: Un flotante *potencial*

```
1 Sea total = 0
2 Sea n = longitud(Cargas)
3 Sea i, j = 1
4 for i = 1, ..., n do
5     for j = i + 1, ..., n do
6         total =
7             total + Cargas[i] * Cargas[j] / norma(Posiciones[i] - Posiciones[j])
8     end
9 end
10 total = total * 1/4πϵ0
11 regresar total
```

Complejidades del algoritmo 3

Hipótesis

- Sumar, multiplicar y dividir dos flotantes toma una cantidad constante de tiempo y es la unidad usada.
- Almacenar un flotante toma una cantidad constante de espacio y es la unidad usada.

Conclusiones

$$T(A) \sim n^2$$

$$E(A) \sim 1$$

Ejemplo real: calcular interacciones electrostáticas

Algoritmo 4: Cálculo de energía electrostática

Input : Un arreglo de vectores de \mathbb{R}^3 *Posiciones*, Un arreglo de flotantes \mathbb{R}^3 *Cargas*

Output: Una matriz de flotantes *V*

```
1 Sea  $n = \text{longitud}(\text{Cargas})$ 
2 Sea  $V$  una matriz de flotantes de  $n \times n$ 
3 Sea  $i, j = 1$ 
4 for  $i = 1, \dots, n$  do
5     for  $j = i + 1, \dots, n$  do
6          $V_{i,j} = +\text{Cargas}[i] * \text{Cargas}[j] / \text{norma}(\text{Posiciones}[i] - \text{Posiciones}[j])^2$ 
7          $V_{i,j} = V_{i,j} * 1/4\pi\epsilon_0$ 
8     end
9 end
```

Complejidades del algoritmo 4

Hipótesis

- Sumar, multiplicar y dividir dos flotantes toma una cantidad constante de tiempo y es la unidad usada.
- Almacenar un flotante toma una cantidad constante de espacio y es la unidad usada.

Conclusiones

$$T(A) \sim n^2$$

$$E(A) \sim n^2$$

Notación Asintótica

Sea $g(n)$ una función positiva y creciente. Denotamos los siguientes conjuntos

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c_2, n_0 \in \mathbb{R}^+ 0 \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

$$\Omega(g(n)) = \{f(n) \mid \exists c_1, n_0 \in \mathbb{R}^+ 0 \leq c_1 g(n) \leq f(n) \forall n \geq n_0\}$$

$$\begin{aligned}\Theta(g(n)) &= \mathcal{O}(g(n)) \cap \Omega(g(n)) \\ &= \{f(n) \mid \exists c_1, c_2, n_0 \in \mathbb{R}^+ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}\end{aligned}$$

Definición visual

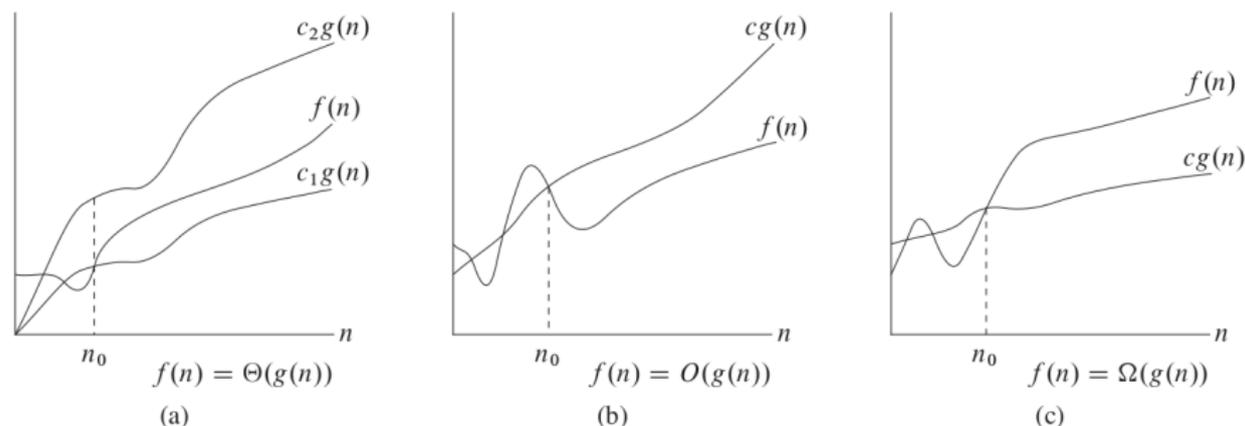


Figura 0.1: Conjunto $\Theta(g(n))$ (a), $\mathcal{O}(g(n))$ (b) y $\Omega(g(n))$ (c) [?]

$\mathcal{O}(g(n))$ es la cota superior asintótica de g , $\Omega(g(n))$ es la cota inferior asintótica de g .

Normalmente expresamos las complejidades como pertenecientes a alguno de estos conjuntos. Si expresamos que pertenecen a $\mathcal{O}(g(n))$, entonces estamos acotando superiormente su complejidad. Es decir, describimos el **peor caso**.

Si expresamos que pertenecen a $\Omega(g(n))$, entonces estamos acotando superiormente su complejidad. Es decir, describimos el **mejor caso**.

Ejemplo canónico: el problema de ordenar

Dado un arreglo A de n números enteros, buscamos un algoritmo que nos regrese un arreglo ordenado C que contenga los mismos números.

Es claro que este problema tiene muchas aplicaciones en la computación con la que interactuamos todos los días.

Solución intuitiva: ir uno por uno

Algoritmo 5: Selection sort

Input : Un arreglo de enteros A

Output: Un arreglo ordenado de enteros C

```
1 Sea  $n = \text{longitud}(A)$ 
2 Sea  $C$  un arreglo de enteros de longitud  $n$ 
3 Sea  $i = 1$ 
4 for  $i = 1, \dots, n$  do
5     //Queremos encontrar el mínimo de  $A[i, \dots, n]$  y colocarlo en  $C[i]$ 
6     Sea  $min = A[i]$  for  $j = i + 1, \dots, n$  do
7         if  $A[j] \geq min$  then
8              $min = A[j]$ 
9         end
10    end
11     $C[i] = min$ 
12 end
13 Regresar  $C$ 
```

Complejidades del algoritmo 5

Suponiendo que hacer comparaciones toma un tiempo constante, hacemos $\sum_{i=1}^n (n - i) = n^2 - \frac{n(n+1)}{2} = \frac{n^2 - n}{2}$ comparaciones.

Así, $T(n) \in \Theta(n^2 - n)$

Debido a la lista que creamos, tenemos $E(n) \in \Theta(n)$

Solución inteligente: separar la lista y hacerlo por partes

Algoritmo 6: Merge sort

Input : Un arreglo de enteros A

Output: Un arreglo ordenado de enteros C

```
1 Sea  $n = \text{longitud}(A)$ 
2 if  $n = 1$  then
3   |   regresar  $A$ 
4 else
5   |   Sea  $c = \lfloor n/2 \rfloor$ 
6   |   Sea  $B = \text{ordenar}(A[1, \dots, c])$ 
7   |   Sea  $C = \text{ordenar}(A[c, \dots, n])$ 
8   |   regresar unirOrdenadamente( $B, C$ )
9 end
```

Solución inteligente: separar la lista y hacerlo por partes

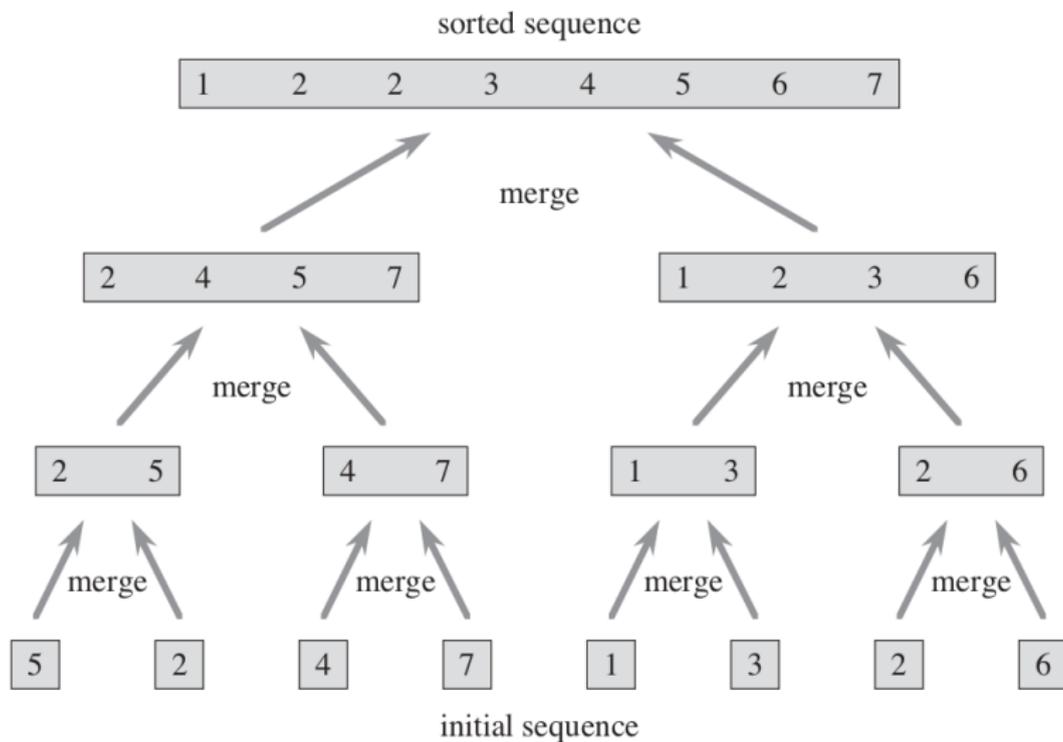


Figura 0.2: Visualización de Merge sort [?]

Complejidades del algoritmo 6

El tiempo de ejecución satisface la relación $T(n) = 2T(n/2) + kn$.

Se puede probar que la solución a esa recurrencia es $T(n) \in \Theta(n \log_2 n)$

Debido a la lista que creamos al final, tenemos $E(n) \in \Theta(n)$

Solución muy inteligente: ver la lista de otra manera

Algoritmo 7: Tournament sort

Input : Un arreglo de enteros A

Output: Un arreglo ordenado de enteros C

- 1 Sea C un arreglo de enteros de longitud n
 - 2 **for** $i = 1, \dots, n$ **do**
 - 3 | Crear un torneo para encontrar el mínimo de $A[i, \dots, n]$
 - 4 | Sea $C[i] = \min(A[i, \dots, n])$
 - 5 **end**
 - 6 regresar C
-

Solución muy inteligente: ver la lista de otra manera

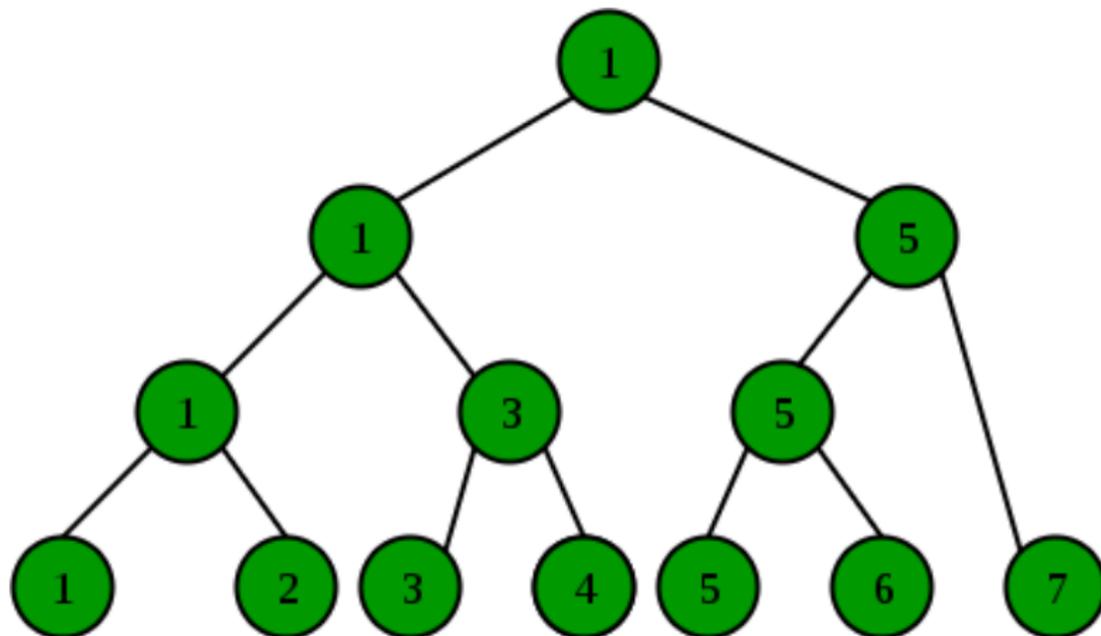


Figura 0.3: Visualización del torneo no uniforme

Complejidades del algoritmo 7

Inicializar el torneo toma $n - 1$ comparaciones. Para quitar un elemento lo debemos de recorrer. El torneo tiene altura $\log_2(n)$ y se recorre n veces.

Concluimos $T(n) \in \Theta(n \log_2 n)$

Debido a la lista que creamos al final, tenemos $E(n) \in \Theta(n)$. Sin embargo, existe una variación (Heapsort) que cumple $E(n) \in \Theta(1)$

Referencias