



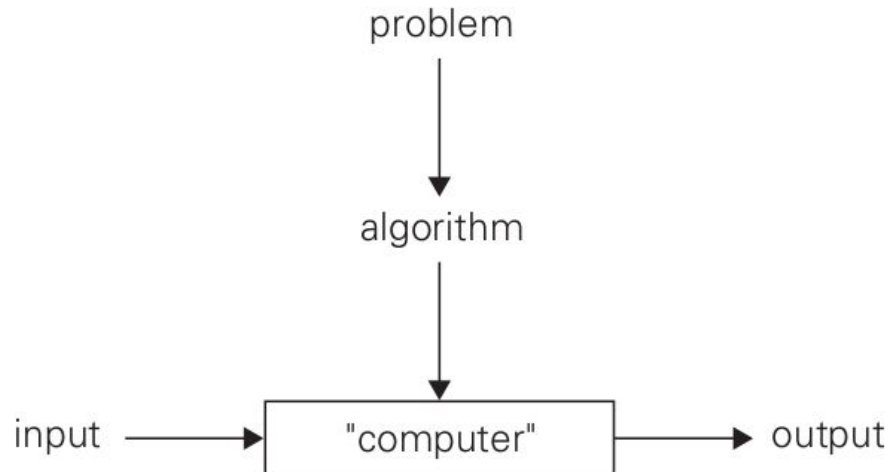
Análisis y diseño de algoritmos

Repaso primer parcial

Dr. Armando Cervantes

¿Que es un algoritmo?

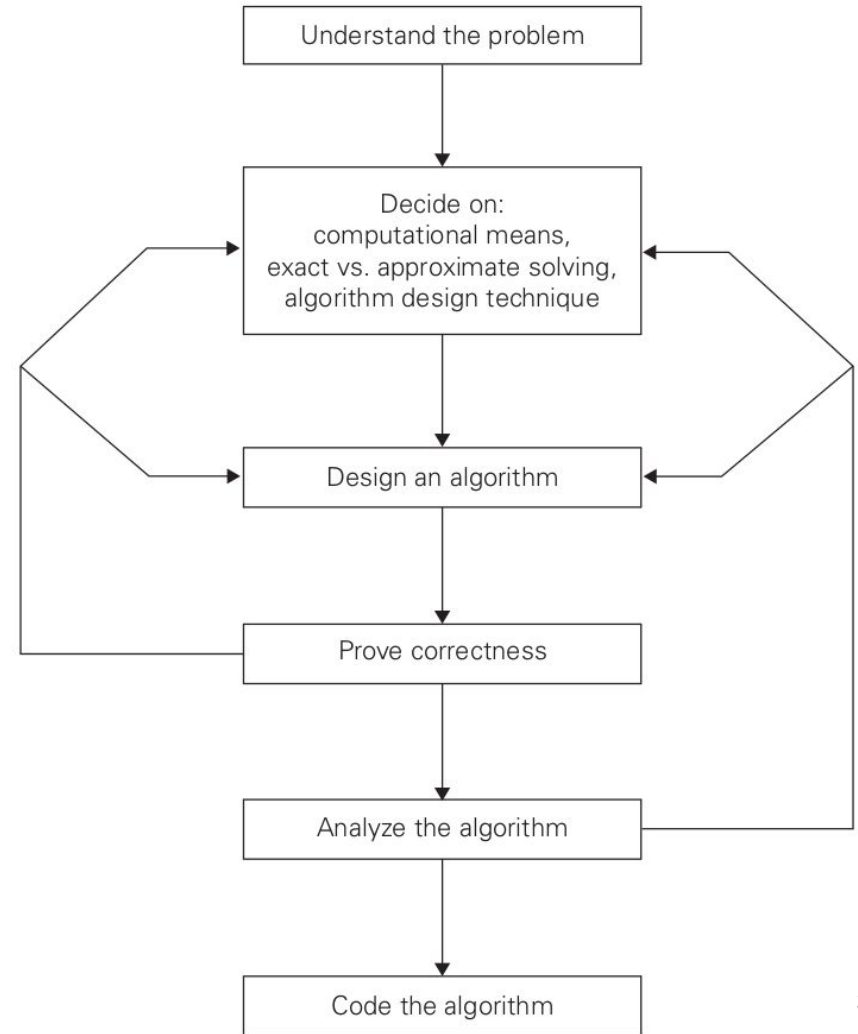
Un algoritmos es una secuencia no ambigua de instrucciones para solucionar un problema, en una cantidad finita de tiempo.



¿Cómo se diseña un algoritmo?

Estructuras de datos fundamentales:

- Listas: arrays, listas ligadas, cadenas
- Stacks
- Colas
- Colas con prioridad
- Grafos
- Árboles
- Diccionarios



Eficiencia Temporal y espacial



- **Eficiencia temporal:** También llamada **complejidad temporal**, nos indica tan rápido corre un algoritmo respecto al tamaño de la entrada.
- **Eficiencia espacial:** También llamada **complejidad espacial**, nos indica que cantidad de memoria es requerida para procesar una tamaño determinado de datos de entrada.

Eligiendo una unidad para medir el tiempo

Medición empírica: Una opción es contar el número de milisegundos que se tarda en resolver un conjunto de entradas de pruebas. Sin embargo esto no es confiable debido a varios factores como:

- La velocidad del equipo en el que corremos el algoritmo.
- El lenguaje
- El compilador utilizado
- Otros procesos que el equipo esté corriendo en ese momento.



Necesitamos una métrica que no dependa en factores externos.

Midiendo el tamaño de la entrada



Un dato evidente es que casi todos los algoritmos tardan más en dar resultados para entradas mas grandes. Para esto se representa el tamaño de la entrada como n .

Por ejemplo para problemas de ordenamiento, búsqueda, obtención de mínimos; n será igual al tamaño de la lista.

Este no siempre es el caso, en ocasiones puede ser el número de bits de una determinada entrada para algoritmos que trabajan sobre operaciones de bits, o el número de caracteres o palabras para algoritmos de procesamiento de texto.

Eligiendo una unidad para medir el tiempo



Podríamos contar el número de operaciones que el algoritmo realiza... Esto puede llegar a ser excesivo e innecesario, en su lugar identificamos la operación más importante de nuestro algoritmo, la más costosa, esta es nuestra **operación básica** y contamos el número de veces que esta es llamada.

Problema	Medida del tamaño de entrada	Operación básica
Buscar un valor en una lista de n elementos	Número de elementos en la lista, n	Comparación de valores
Problema típico de grafos	Número de vértices y/o arcos	Visita a un vértice o recorrido de un arco

Notación asintótica



La notación asintótica es conocida como la Gran-O, o en inglés big-O notation.

- Es denotado como $O(g(n))$
- De ahí su nombre
- Al comparar funciones ignora factores constantes

Cuando nos referimos a $O(g(n))$ son TODAS las clases de funciones $f(n)$ que no crecen más rápido que $g(n)$.

Ejemplos de big-O



$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n-1) \in O(n^2).$$

$$t(n) \leq cg(n), \quad \forall n \geq n_0$$

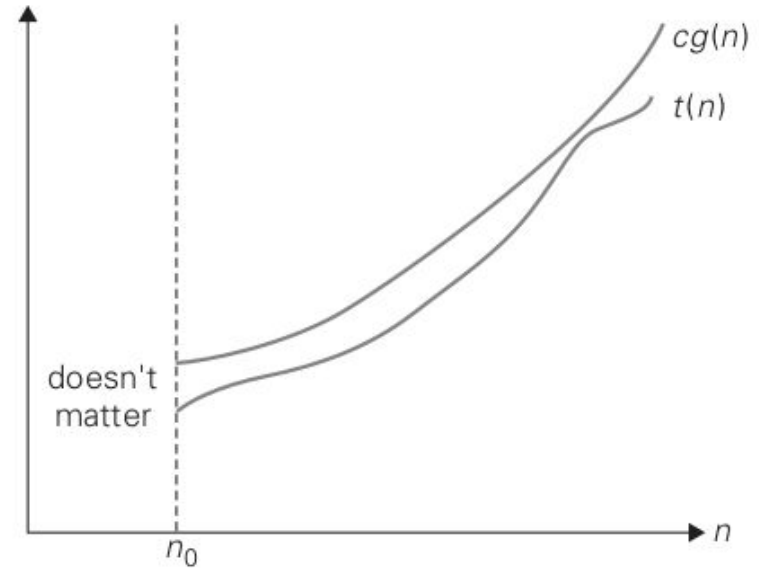


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

Clases básicas de eficiencia asintótica



Clase	Nombre
1	constante
$\log n$	logarítmico
n	lineal
$n \log n$	n-log-n o linealítmico
n^2	cuadrático
n^3	cúbico
2^n	exponencial
$n!$	factorial


Análisis matemático de algoritmos iterativos



Propiedades y reglas para sumatorias:

- Existen muchas propiedades e identidades de las sumatorias
- Lo importante no es memorizarlas, si no comprenderlas
- Tanto el libro de Levitin como el de Cormen cuentan con Apéndices con este tipo de información

Reglas y propiedades para sumatorias - 1


$$\sum_{i=l}^u 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ veces}} = u - l + 1 \quad (l, u \text{ son enteros y } l \leq u)$$

En particular

$$\sum_{i=1}^n 1 = n - 1 + 1 = n \in \Theta(n)$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=3}^{n+1} 1 = (n+1) - 3 + 1 = n - 1$$

Reglas y propiedades para sumatorias - 2




$$\sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=3}^{n+1} i = \sum_{i=0}^{n+1} i - \sum_{i=0}^2 i = \frac{(n+1)(n+2)}{2} - 3 = \frac{n^2 + 3n - 4}{2}$$

Reglas y propiedades para sumatorias - 3


$$\sum_{i=1}^n i^2 = 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=0}^{n-1} i^2 = 0 + \sum_{i=1}^{n-1} i^2 = \frac{(n-1)(n-1+1)(2(n-1)+1)}{6} = \frac{(n-1)(n)(2n-1)}{6}$$

Reglas y propiedades para sumatorias - 4



$$\sum_{i=0}^n a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a \neq 1)$$

En particular

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1 \in \Theta(2^n)$$

Ejercicio, calcule la siguiente sumatoria:

$$\sum_{i=0}^n 5^i = \frac{5^{(n)+1} - 1}{5 - 1} = \frac{5^{n+1} - 1}{4}$$

Reglas y propiedades para sumatorias - 5



$$\sum_{i=1}^n \log i \approx n \log n$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=2}^{n-1} 2 \log i = 2 \sum_{i=2}^{n-1} \log i = 2 \sum_{i=1}^n \log i - 2 \log n = 2(n \log n) - 2 \log n$$

Reglas y propiedades para sumatorias - 6



$$\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=2}^{n-1} 8i^4 = 8 \sum_{i=2}^{n-1} i^4$$

Reglas y propiedades para sumatorias - 7




$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=0}^{n-1} (i^4 + i^2 - 3) = \sum_{i=0}^{n-1} i^4 + \sum_{i=0}^{n-1} i^2 - \sum_{i=0}^{n-1} 3$$

Reglas y propiedades para sumatorias - 8


$$\sum_{i=l}^u a_i = \sum_{i=l}^m a_i + \sum_{i=m+1}^u a_i \quad (l \leq m < u)$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=0}^n (i^2 + 1)^2 = \sum_{i=0}^{\frac{n}{3}} (i^2 + 1)^2 + \sum_{i=\frac{n}{3}+1}^{\frac{2n}{3}} (i^2 + 1)^2 + \sum_{i=\frac{2n}{3}+1}^n (i^2 + 1)^2$$

Reglas y propiedades para sumatorias - 10



$$\sum_{i=1}^n \sum_{j=1}^n c = \sum_{i=1}^n \underbrace{\left(c + c + \cdots + c \right)}_{n \text{ veces}} = \sum_{i=1}^n nc = \underbrace{\left(nc + nc + \cdots + nc \right)}_{n \text{ veces}} = n(nc) = n^2 c$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=1}^n \sum_{j=1}^n 5 = n^2(5)$$

Reglas y propiedades para sumatorias - 11



$$\sum_{i=1}^n \sum_{j=1}^n \left(bx_{ij} + c \right) = \sum_{i=1}^n \sum_{j=1}^n bx_{ij} + \sum_{i=1}^n \sum_{j=1}^n c$$

Ejemplo, calcule la siguiente sumatoria:

$$\sum_{i=1}^n \sum_{j=1}^n \left((i + j) + 4 \right) = \sum_{i=1}^n \sum_{j=1}^n (i + j) + \sum_{i=1}^n \sum_{j=1}^n 4$$

Análisis de algoritmos iterativos



Plan general:

1. Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo
2. Identificar la operación básica del algoritmo
3. Determinar los casos peor, promedio y mejor para una entrada de tamaño n
4. Expresar como una sumatoria el número de veces que la operación básica es ejecutada por el algoritmo analizado
5. Simplificar la sumatoria empleando propiedades y reglas estándar

Análisis de algoritmos iterativos

ALGORITHM *MaxElement*($A[0..n-1]$)

//Determines the value of the largest element in a given array

//Input: An array $A[0..n-1]$ of real numbers

//Output: The value of the largest element in A

$maxval \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

if $A[i] > maxval$

$maxval \leftarrow A[i]$

return $maxval$

1. Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo: **n es el tamaño del arreglo**
2. Identificar la operación básica del algoritmo: **comparaciones del algoritmo**
3. Determinar los casos peor, promedio y mejor para una entrada de tamaño n : **No existen en este caso**
4. Expresar como una sumatoria el número de veces que la operación básica es ejecutada por el algoritmo analizado: $\sum_{i=1}^{n-1} 1$
5. Simplificar la sumatoria empleando propiedades y reglas estándar


$$\sum_{i=1}^{n-1} 1 = n - 1 \in O(n)$$

ALGORITHM *UniqueElements*($A[0..n - 1]$)

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n - 1]$
//Output: Returns “true” if all the elements in A are distinct
// and “false” otherwise
for $i \leftarrow 0$ **to** $n - 2$ **do**
 for $j \leftarrow i + 1$ **to** $n - 1$ **do**
 if $A[i] = A[j]$ **return false**
return true

1. Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo: **n es el tamaño del arreglo**
2. Identificar la operación básica del algoritmo: **comparaciones del algoritmo**
3. Determinar los casos peor, promedio y mejor para una entrada de tamaño n : **sólo analizaremos el peor caso (arreglo sin elementos repetidos o últimos 2 elementos son el único par repetido)**
4. Expresar como una sumatoria el número de veces que la operación básica es ejecutada por el algoritmo analizado:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$



ALGORITHM *MatrixMultiplication*($A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$)
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n - 1$ **do**
 for $j \leftarrow 0$ **to** $n - 1$ **do**
 $C[i, j] \leftarrow 0.0$
 for $k \leftarrow 0$ **to** $n - 1$ **do**
 $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

1. Decidir acerca del parámetro n que indica el tamaño de la entrada del algoritmo: **n es el tamaño de la matriz.**
2. Identificar la operación básica del algoritmo: **la multiplicación**
3. Determinar los casos peor, promedio y mejor para una entrada de tamaño n : **no existen en este caso**
4. Expresar como una sumatoria el número de veces que la operación básica es ejecutada:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Análisis de algoritmos iterativos



5. Simplificar la sumatoria empleando propiedades y reglas estándar

$$\begin{aligned}M(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 \\&= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n \\&= \sum_{i=0}^{n-1} n^2 \\&= n^3\end{aligned}$$

Teorema Maestro



Si tenemos un algoritmo recursivo que puede ser expresado como:

$$T(n) = aT(n/b) + f(n)$$

a = Número de llamados recursivos

b = División del tamaño original de cada llamada

$f(n)$ = el costo de partir y juntar los datos en cada llamado que debe

cumplir: $f(n) \in \Theta(n^d)$

Teorema Maestro

Si tenemos un algoritmo recursivo que puede ser expresado como:

$$T(n) = aT(n/b) + f(n)$$

a = Número de llamados recursivos

b = División del tamaño original de cada llamada

$f(n)$ = el costo de partir y juntar los datos en cada llamado que debe cumplir:
 $f(n) \in \Theta(n^d)$

MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r)/2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

Teorema Maestro

Si tenemos un algoritmo recursivo que puede ser expresado como:

$$T(n) = aT(n/b) + f(n)$$

a = Número de llamados recursivos

b = División del tamaño original de cada llamada

f(n)= el costo de partir y juntar los datos en cada llamado que debe

cumplir: $f(n) \in \Theta(n^d)$

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

Teorema Maestro



Si tenemos un algoritmo recursivo que puede ser expresado como:

$$T(n) = aT(n/b) + f(n)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{si } a < b^d \\ \Theta(n^d \log n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Teorema Maestro: Ejemplo



MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

El número de subproblemas es 2 (línea 3 y 4), por lo tanto $a = 2$

Cada subproblema lo divide a la mitad $T(n/2)$ es decir $b = 2$

El costo de juntar los datos en cada llamado es n para que $n^d = n$, $d = 1$

Teorema Maestro: Ejemplo



MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

El número de subproblemas es 2 (línea 3 y 4), por lo tanto $a = 2$

Cada subproblema lo divide a la mitad $T(n/2)$ es decir $b = 2$

El costo de juntar los datos en cada llamado es n para que $n^d = n$, $d = 1$

$$a = 2$$

$$b^d = 2^1 = 2$$

Teorema Maestro: Ejemplo

MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

El número de subproblemas es 2 (línea 3 y 4), por lo tanto $a = 2$

Cada subproblema lo divide a la mitad $T(n/2)$ es decir $b = 2$

El costo de juntar los datos en cada llamado es n para que $n^d = n$, $d = 1$

$a = 2$

$b^d = 2^1 = 2$

Tenemos que $a = b^d$

$$T(n) = \begin{cases} \Theta(n^d) & \text{si } a < b^d \\ \Theta(n^d \log n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Teorema Maestro: Ejemplo

MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

El número de subproblemas es 2 (línea 3 y 4), por lo tanto $a = 2$

Cada subproblema lo divide a la mitad $T(n/2)$ es decir $b = 2$

El costo de juntar los datos en cada llamado es n para que $n^d = n$, $d = 1$

$$a = 2$$

$$b^d = 2^1 = 2$$

$$\Theta(n^d \log n) \quad \text{si } a = b^d$$

Tenemos que $a = b$, por lo tanto tenemos el caso 2 del teorema maestro!

Teorema Maestro: Ejemplo

MERGE-SORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

El número de subproblemas es 2 (línea 3 y 4), por lo tanto $a = 2$

Cada subproblema lo divide a la mitad $T(n/2)$ es decir $b = 2$


El costo de juntar los datos en cada llamado es n para que $n^d = n$, $d = 1$

$$a = 2$$
$$b^d = 2^1 = 2$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log n)$$


Tenemos que $a = b$, por lo tanto tenemos el caso 2 del teorema maestro!

Teorema Maestro: Ejemplo hipotético recursión pesada



Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a sí mismo 8 veces en cada recursión!!!!

Teorema Maestro: Ejemplo hipotético recursión pesada



Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a sí mismo 8 veces en cada recursión!!!!

- Identificamos que para esta recurrencia $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, por lo tanto $d = 2$

Teorema Maestro: Ejemplo hipotético recursión pesada

Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a si mismo 8 veces en cada recursión!!!!

- Identificamos que para esta recurrencia $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, por lo tanto $d = 2$

$$\begin{array}{l} a = 8 \\ b^d = 2^2 = 4 \end{array} \quad T(n) = \begin{cases} \Theta(n^d) & \text{si } a < b^d \\ \Theta(n^d \log n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Tenemos que $a > b^d$

Teorema Maestro: Ejemplo hipotético recursión pesada

Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a si mismo 8 veces en cada recursión!!!!

- Identificamos que para esta recurrencia $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, por lo tanto $d = 2$

$$a = 8$$

$$b^d = 2^2 = 4$$

$$\Theta(n^{\log_b a}) \quad \text{si } a > b^d$$

Tenemos que $a > b$, por lo tanto tenemos el caso 3 del teorema maestro!

Teorema Maestro: Ejemplo hipotético recursión pesada

Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a si mismo 8 veces en cada recursión!!!!

- Identificamos que para esta recurrencia $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, por lo tanto $d = 2$

$$a = 8$$

$$b^d = 2^2 = 4$$

$$\Theta(n^{\log_b a}) \quad \text{si } a > b^d$$

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Teorema Maestro: Ejemplo hipotético recursión pesada

Imaginemos un algoritmo, que en cada recursión, divide la data a la mitad, y le toma n^2 en juntar los datos, sin embargo se llama a si mismo 8 veces en cada recursión!!!!

- Identificamos que para esta recurrencia $a = 8$, $b = 2$, $f(n) = \Theta(n^2)$, por lo tanto $d = 2$

$$a = 8$$

$$b^d = 2^2 = 4$$

$$\Theta(n^{\log_b a}) \quad \text{si } a > b^d$$

$$T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^{\log_2(8)}) = \Theta(n^3)$$

Selection sort


input : Un arreglo A de tamaño n

output: El arreglo A ordenado

```
1 for  $i \leftarrow 0$ ;  $i < n - 1$ ;  $i++$  do
    /* Encontrar el mínimo elemento en el arreglo desordenado */
2      $minIndex \leftarrow i$ ;
3     for  $j \leftarrow i + 1$ ;  $j < n$ ;  $j++$  do
4         if  $A[j] < A[minIndex]$  then  $minIndex \leftarrow j$ ;
5     end
    /* Intercambiar el mínimo elemento encontrado con el primero */
6      $temp \leftarrow A[minIndex]$ ;
7      $A[minIndex] \leftarrow A[i]$ ;
8      $A[i] \leftarrow temp$ ;
9 end
```

¿Qué complejidad temporal tiene?

Selection sort


$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

Bubble sort


input : Un arreglo A de tamaño n

output: El arreglo A ordenado

```
1 for  $i \leftarrow 0$ ;  $i < n - 1$ ;  $i++$  do
2   for  $j \leftarrow 0$ ;  $j < n - i - 1$ ;  $j++$  do
3     if  $A[j] > A[j + 1]$  then
4       /* Intercambiar  $A[j + 1]$  y  $A[j]$  */
5        $temp \leftarrow A[j]$ ;
6        $A[j] \leftarrow A[j + 1]$ ;
7        $A[j + 1] \leftarrow temp$ ;
8     end
9   end
10 end
```

¿Qué complejidad temporal tiene?

Bubble sort


$$\sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{(n-1)n}{2} \in \Theta(n^2).$$

Insertion sort

input : Un arreglo A de tamaño n

output: El arreglo A ordenado

```
1 for  $i \leftarrow 1$ ;  $i < n$ ;  $++i$  do
2    $key \leftarrow A[i]$ ;
3    $j \leftarrow i - 1$ ;
   // Mover elementos  $A[0..j]$ , que son
   // mayores a  $key$ , una posición adelante
4   while  $j \geq 0$  and  $A[j] > key$  do
5      $A[j + 1] \leftarrow A[j]$ ;
6      $j \leftarrow j - 1$ 
7   end
8    $A[j + 1] \leftarrow key$ ;
9 end
```

Insertion sort

input : Un arreglo A de tamaño n

output: El arreglo A ordenado

```
1 for  $i \leftarrow 1$ ;  $i < n$ ;  $++i$  do
2    $key \leftarrow A[i]$ ;
3    $j \leftarrow i - 1$ ;
   // Mover elementos  $A[0..j]$ , que son
   // mayores a  $key$ , una posición adelante
4   while  $j \geq 0$  and  $A[j] > key$  do
5      $A[j + 1] \leftarrow A[j]$ ;
6      $j \leftarrow j - 1$ 
7   end
8    $A[j + 1] \leftarrow key$ ;
9 end
```

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Tablas de complejidad



Algoritmo	Mejor	Promedio	Peor	Estable	Espacio
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Problema



Imagine que tienen un arreglo de 200 millones de registros con edades, ¿Cuál sería el mejor algoritmo de ordenamiento que podrían usar?

¿Cuánta memoria consumen?

Counting sort



COUNTING-SORT(A, B, k)

```
1  let  $C[0 \dots k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

¿Cual es la complejidad temporal de este algoritmo?

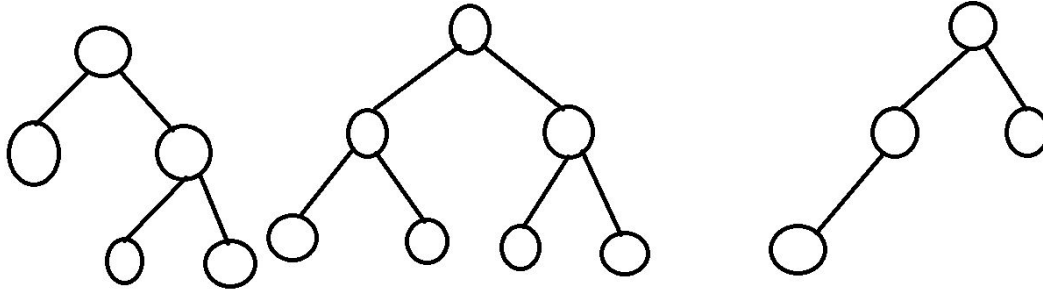
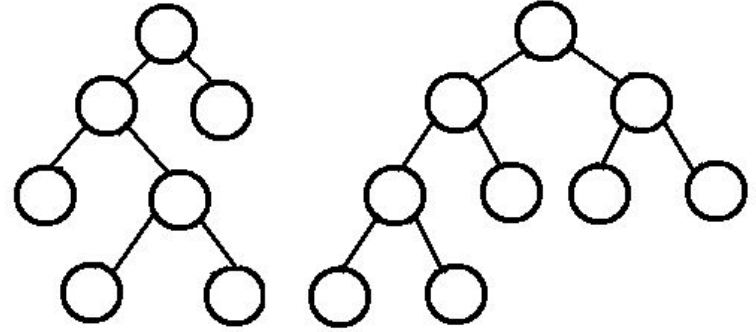
$n + k$

<https://www.youtube.com/watch?v=7zuGmKfUt7s>

Árbol lleno vs Arbol completo

Un árbol lleno, es un árbol donde cada nodo tiene 0 o dos hijos.

Un árbol completo es un árbol completamente lleno con la posible excepción de su último nivel el cual estará llenado lo mayormente posible, de izquierda a derecha.

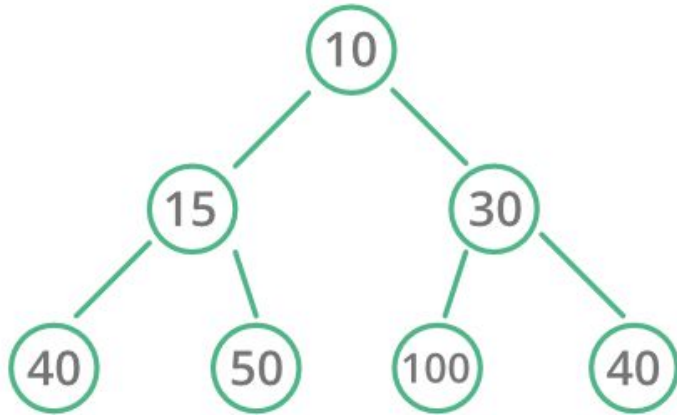


Cual es la altura máxima (respecto a n) de estos árboles?

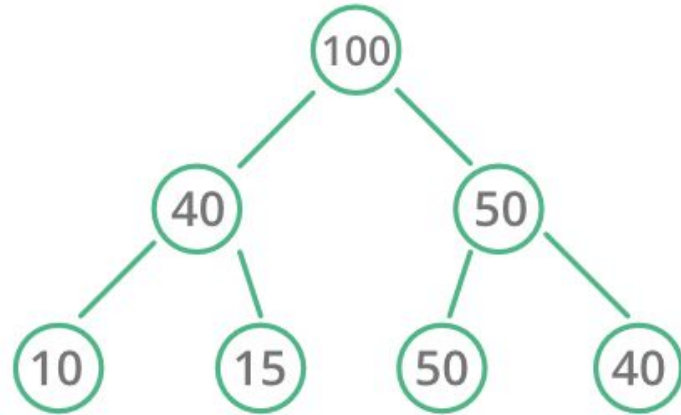
Heap

Es un árbol binario completo que cumple con una condición de heap:

- a) Min heap: El valor del padre siempre es menor que el hijo
- b) Max heap: El valor del padre siempre es mayor que el del hijo



Min Heap



Max Heap

Heap sort



HEAPSORT(A)

https://www.youtube.com/watch?v=MtQL_I15KhQ

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

Explicacion detallada <https://www.youtube.com/watch?v=HqPJF2L5h9U>

Heap sort

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.\text{length}$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.\text{heap-size} = A.\text{heap-size} - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

Heap sort



HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

BUILD-MAX-HEAP(A)

```
1   $A.heap-size = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )
```

Heap sort



HEAPSORT(A)

$O(n \log n)$

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```


Radix sort

101	1	20	50	9	98	27	153	35	899
-----	---	----	----	---	----	----	-----	----	-----



0	1	2	3	4	5	6	7	8	9
20	101		153		35		27	98	9
50	1								899



20	50	101	1	153	35	27	98	9	899
----	----	-----	---	-----	----	----	----	---	-----

Radix sort

20	50	101	1	153	35	27	98	9	899
----	----	-----	---	-----	----	----	----	---	-----



0	1	2	3	4	5	6	7	8	9
101		20	35		50				98
01		27			153				899
09									



101	1	9	20	27	35	50	153	98	899
-----	---	---	----	----	----	----	-----	----	-----

Radix sort



20	50	101	1	153	35	27	98	9	899
----	----	-----	---	-----	----	----	----	---	-----



0	1	2	3	4	5	6	7	8	9
001	101							899	
009	153								
020									
027									
035									
050									
098									



1	9	20	27	35	50	98	101	153	899
---	---	----	----	----	----	----	-----	-----	-----

Radix sort



RadixSort(A, d)

 for i = 1 to d

 kindOfBucketSortOnDigit(A,i)

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

Hash tables



Una tabla hash es una estructura de datos que almacena información de una manera asociativa.

Es decir, cada valor almacenado dentro de nuestra tabla tiene asociada una “llave” (key) que usamos para buscar su determinar su posición.

¿Que hace tan especiales a las hash table?

Su complejidad de búsqueda e inserción son constantes! (en promedio)

Hash tables



¿Cómo se logra esto? Se implementa una función que con los elementos en integers (llaves hash, valores hash) utilizados como el índice de búsqueda en nuestra tabla.

¿Que necesita una buena función hash?

- Fácil de calcular: El costo de convertir un elemento en una llave hash no debe de ser grande.
- **Uniformemente distribuido:** Debe de distribuir los elementos convertidos en integers que no se acumulen en una parte del array.
 - $\text{key}(e) = e \% \text{len}(\text{table})$
 - $\text{key}(e) = \max(\text{len}(\text{table}) - 1, e)$ ❌
- Minimizar colisiones: Evitar que diferentes elementos generen el mismo valor hash.
 - Nota: Esto siempre ocurrirá con una cantidad de datos suficientemente grande, nuestro trabajo es tratar de minimizarlo.

Colisiones



Imaginen que tenemos una hash function que transforma string a int sumando el valor de los caracteres ASCII y realizando la operación módulo sobre el resultado. ¿Que pasaria con los siguiente valores?

- {“abcdef”, “bcdefa”, “cdefab” , “defabc” }

Colisiones

Imaginen que tenemos una hash function que transforma string a int sumando el valor de los caracteres ASCII y realizando la operación módulo sobre el resultado. ¿Que pasaria con los siguiente valores?

- {“abcdef”, “bcdefa”, “cdefab”, “defabc” }

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

¿Cómo podemos corregir esto?

Colisiones

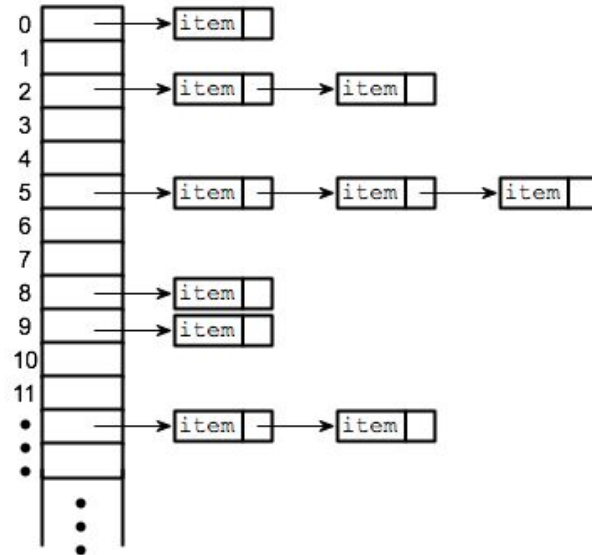
Podríamos tomar el valor de los caracteres ASCII, multiplicarlo por 10 y sumarle su posición en la cadena, sumar todos y posteriormente realizar la operación módulo sobre el resultado.

String	Hash function	Index
abcdef	$(971 + 982 + 993 + 1004 + 1015 + 1026)\%2069$	38
bcdefa	$(981 + 992 + 1003 + 1014 + 1025 + 976)\%2069$	23
cdefab	$(991 + 1002 + 1013 + 1024 + 975 + 986)\%2069$	14
defabc	$(1001 + 1012 + 1023 + 974 + 985 + 996)\%2069$	11

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Resolviendo colisiones: Open hashing

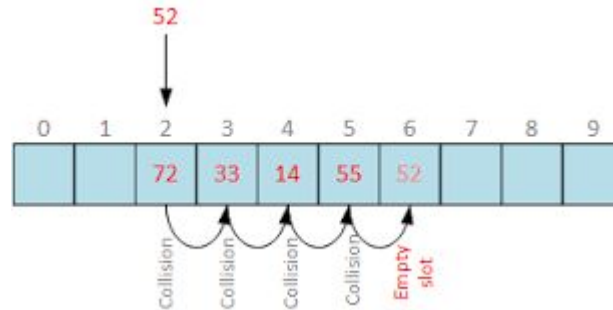
Una posible solución es convertir nuestro array de strings en un array de listas ligadas (buckets). Cada vez que exista una colisión (se quiera escribir datos en una posición que ya contiene datos), el elemento se agrega a la lista ligada



Resolviendo colisiones: Open addressing

Open addressing (close hashing, linear probing) nos permite seguir utilizando un array de integers (sin buckets). Cada vez que existe una colisión, nuestra tabla inserta nuestro elemento en el **próximo slot vacío** de la tabla.

En caso de llegar al final de la tabla, el próximo slot se considera el 0.



<https://www.youtube.com/watch?v=1maxi3Yq-CU>

Factor de carga y rehashing



Una tabla hash puede con open addressing puede ser llenada. ¿Cómo resolvemos esto?

Consideremos el porcentaje en el que una tabla se encuentra ocupada, esto es llamado factor de carga (**Load factor**), cuando el factor de carga sobrepasa un umbral (0.75 en varias implementaciones), se procede a lo siguiente:

- El tamaño de la tabla se dobla
- Todos los elementos en la hash table son calculados de nuevo “Rehashing”

<https://www.youtube.com/watch?v=7PuZOsxe-mo>