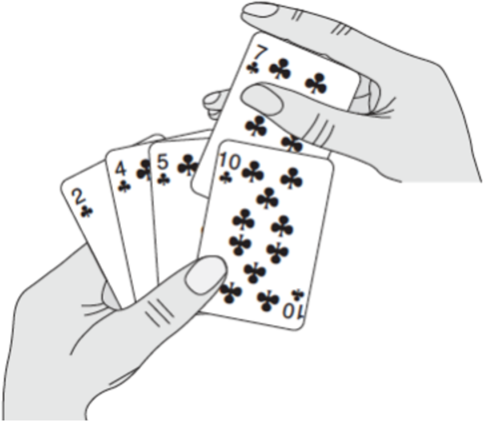
Fundamentos de Diseño de Algoritmos

Marks Calderón Niquin

Universidad ESAN

Temario

Tiempo de jugar



https://youtu.be/K4CuPzdiAIo

Algoritmo de Inserción

Entrada: Una secuencia de n números $\langle a_1, a_2, ..., a_n \rangle$.

Salida: Una permutación (reordenamiento) $\left\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \right\rangle$

Algoritmo de Inserción

- Los números a ordenar se denominarán keys.
- El algoritmo simula ordenar un mazo de cartas.

Entrada: Una secuencia de n números $\langle a_1, a_2, ..., a_n \rangle$.

Salida: Una permutación (reordenamiento) $\left\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \right\rangle$

Algoritmo de Inserción

- Los números a ordenar se denominarán **keys**.
- El algoritmo simula ordenar un mazo de cartas.

Algoritmo de Inserción

Entrada: Una secuencia de n números $\langle a_1, a_2, ..., a_n \rangle$.

Salida: Una permutación (reordenamiento) $\left\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \right\rangle$

Algoritmo de Inserción

- Los números a ordenar se denominarán **keys**.
- El algoritmo simula ordenar un mazo de cartas.

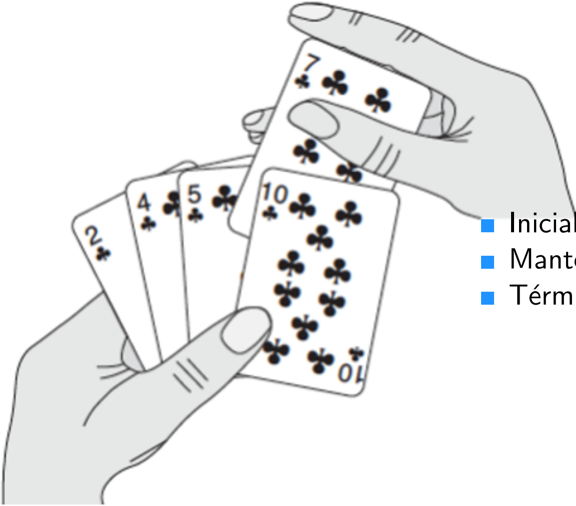
Entrada: Una secuencia de n números $\langle a_1, a_2, ..., a_n \rangle$.

Salida: Una permutación (reordenamiento) $\left\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \right\rangle$

Algoritmo de Inserción

- Los números a ordenar se denominarán **keys**.
- El algoritmo simula ordenar un mazo de cartas.

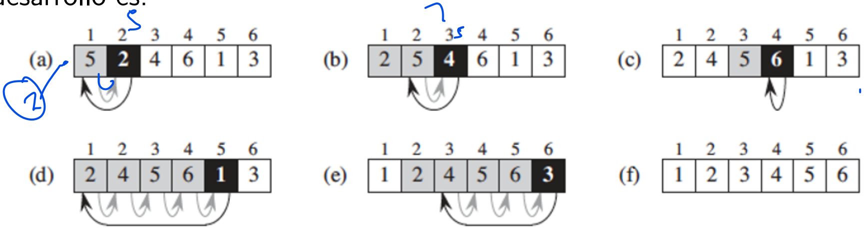
Algoritmos de Inserción



- Inicialización.
- Mantenimiento
- Término

Algoritmo de Inserción

Dado $A =< 5, 2, 4, 6, 1, 3 >$, al aplicar el algoritmo INSERT-SORT su desarrollo es:



$J = n+1$

for $j \leftarrow 2$ **to** $A.length$ **do**

n

  $key = A[j]$

  //insertar $A[j]$en la secuencia

  //ordenada $A[1..j-1]$

  $i = j - 1$

  while

   $i > 0 \&\& A[i] > key$ **do**

    $A[i+1] = A[i]$ desplazar

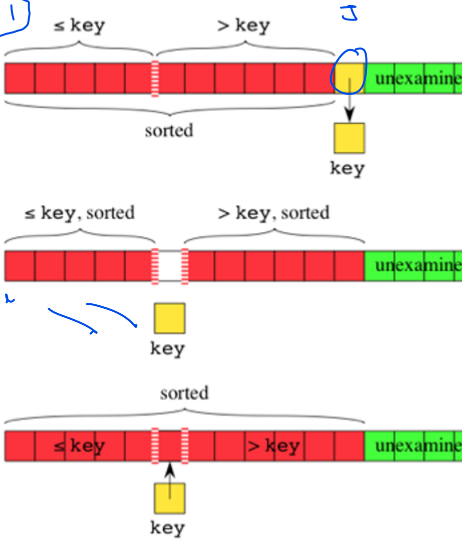
    $i = i - 1$

  end

  $A[i+1] = key$

end

Algorithm: Insertion-Sort(A)



Correctitud de la Inserción

Inicialización: Antes de la primera iteración

- $j = 2$
- El subarreglo $A[1, .., j-1] =$ elemento en $A[1]$
- $A[1]$ está ordenado.

Mantenimiento

- El bucle **for** trabaja movimiendo $A[j-1], A[j-2], A[j-3]$, hasta ubicar la posición apropiada para $A[j]$.
- $A[j]$ se inserta manteniendo el orden.

Terminación

- El bucle **for** finaliza cuando $j > n (j == n+1)$
- Al reemplazar n+1 en j del invariante del bucle, entonces el subarreglo $A[1, .., n]$ esta ordenado

Correctitud de la Inserción

Inicialización: Antes de la primera iteración

- $j = 2$
- El subarreglo $A[1, .., j-1]$ = elemento en $A[1]$
- $A[1]$ está ordenado.

Mantenimiento

- El bucle **for** trabaja movimiendo $A[j-1], A[j-2], A[j-3]$, hasta ubicar la posición apropiada para $A[j]$.
- $A[j]$ se inserta manteniendo el orden.

Terminación

- El bucle **for** finaliza cuando $j > n (j == n+1)$
- Al reemplazar n+1 en j del invariante del bucle, entonces el subarreglo $A[1, .., n]$ esta ordenado

Correctitud de la Inserción

Inicialización: Antes de la primera iteración

- $j = 2$
- El subarreglo $A[1, .., j-1]$ = elemento en $A[1]$
- $A[1]$ está ordenado.

Mantenimiento

- El bucle **for** trabaja movimiendo $A[j-1], A[j-2], A[j-3]$, hasta ubicar la posición apropiada para $A[j]$.
- $A[j]$ se inserta manteniendo el orden.

Terminación

- El bucle **for** finaliza cuando $j > n (j == n+1)$
- Al reemplazar n+1 en j del invariante del bucle, entonces el subarreglo $A[1, .., n]$ esta ordenado

Análisis de Algoritmos

Análisis de algoritmos llegan a predecir los recursos que los algoritmos requieren.

- **Recursos:** memoria, ancho de banda, o hardware de computadora.
- Se analizan diferentes propuestas para escoger la mejor
- Para analizar un algoritmo, debemos tener un modelo de la tecnología de implementación a usar. Por lo general se emplea **RAM(Random-acces machine).**

Análisis de Algoritmos

Análisis de algoritmos llegan a predecir los recursos que los algoritmos requieren.

- **Recursos:** memoria, ancho de banda, o hardware de computadora.
- Se analizan diferentes propuestas para escoger la mejor
- Para analizar un algoritmo, debemos tener un modelo de la tecnología de implementación a usar. Por lo general se emplea **RAM(Random-acces machine).**

Análisis de algoritmos llegan a predecir los recursos que los algoritmos requieren.

- **Recursos:** memoria, ancho de banda, o hardware de computadora.
- Se analizan diferentes propuestas para escoger la mejor
- Para analizar un algoritmo, debemos tener un modelo de la tecnología de implementación a usar. Por lo general se emplea **RAM(Random-acces machine).**

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Random-access memory (RAM)

- Modelo de computación abstracto para ejecutar algoritmos.
- RAM tiene un solo procesador.
- Memoria infinita
- Las instrucciones son ejecutadas consecutivas.
- Además cumple con:
  - Cada operación toma un paso.
  - Bucles y subrutinas no son simples operaciones
  - Cada acceso a memoria toma un paso.

Tipos de Análisis

Peor caso(Worst case)

- $T(n)$: tiempo máximo del algoritmo.
- Cota superior de tiempo de ejecución dada cualquier entrada, asegura que ninguna entrada tomará más del tiempo indicado.

Caso promedio(Average case)

- $T(n)$: tiempo promedio del algoritmo.
- Se asume ciertas características sobre la distribución de los datos.

Mejor Caso(Best-case)

- Solo funciona en muy pocos casos, depende de la entrada de datos del algoritmo.

Tipos de Análisis

Peor caso(Worst case)

- $T(n)$: tiempo máximo del algoritmo.
- Cota superior de tiempo de ejecución dada cualquier entrada, asegura que ninguna entrada tomará más del tiempo indicado.

Caso promedio(Average case)

- $T(n)$: tiempo promedio del algoritmo.
- Se asume ciertas características sobre la distribución de los datos.

Mejor Caso(Best-case)

- Solo funciona en muy pocos casos, depende de la entrada de datos del algoritmo.

Tipos de Análisis

Peor caso(Worst case)

- $T(n)$: tiempo máximo del algoritmo.
- Cota superior de tiempo de ejecución dada cualquier entrada, asegura que ninguna entrada tomará más del tiempo indicado.

Caso promedio(Average case)

- $T(n)$: tiempo promedio del algoritmo.
- Se asume ciertas características sobre la distribución de los datos.

Mejor Caso(Best-case)

- Solo funciona en muy pocos casos, depende de la entrada de datos del algoritmo.

Análisis: Complejidad del algoritmo de Inserción

6

```
for j ← 2 to A.length do
    key = A[j] ;
    //insertar A[j]en la secuencia ordenada A[1..j − 1] ;
    i = j − 1;
    while  i > 0&&A[i] > key do
        A[i + 1] = A[i];
        i = i − 1;
    end
    A[i + 1] = key;
end
```

	Costo	Tiempo
for $j \leftarrow 2$ to $A.length$ do	c_1	n
$key = A[j]$	c_2	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0 \&\& A[i] > key$ do	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Algorithm: Insertion-Sort(A)

- El cálculo de tiempo de ejecución del algoritmo T(n), es la suma de todos los tiempos de cada instrucción ejecutada.
- Si una instrucción se ejecuta **n** este contribuye con **cn** .
- El **T(n)** será:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Análisis: Complejidad del algoritmo de Inserción

$2 \; . \; . \; . \; n$

El $\mathbf{T(n)}$ para el peor caso:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Desarrollando las ecuaciones

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

Finalmente

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

Análisis: Complejidad del algoritmo de Inserción

1 2 3 4 5 6 7

El **T(n)** para el mejor caso:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} \overset{1}{t_j} + c_6 \overset{0}{\cancel{\sum_{j=2}^{n}(t_j - 1)}} + c_7 \underset{0}{\cancel{\sum_{j=2}^{n}(t_j - 1)}} + c_8(n-1)$$

Desarrollando las ecuaciones

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

Finalmente

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$$

Análisis: Complejidad del algoritmo de Inserción

El **T(n)** para el caso promedio:

- Si elegimos aleatoriamente n números y aplicamos INSERTION-SORT.
- En promedio, se verificará la mitad del subarreglo, tal que tj= j/2
- Se obtendrá una función cuadrática, al igual que en el PEOR CASO.
- Para este análisis a veces se necesita un análisis probabilitisco.

Tasa de crecimiento Θ

- Se prioriza el término de mayor orden de la fórmula, sin incluir su coeficiente.
- Se asume que los términos de menor orden y constantes son insignificantes. Por ejemplo: $an^2 + bn + c$, sera n^2.
- Para el caso del algoritmo de ordenación de inserción presenta un tiempo de ejecución $\Theta(n^2)$.
- Un algoritmo es mejor que otro al comparar su tiempo de ejecución en el peor caso.

Tasa de crecimiento Θ

- Se prioriza el término de mayor orden de la fórmula, sin incluir su coeficiente.
- Se asume que los términos de menor orden y constantes son insignificantes. Por ejemplo: $an^2 + bn + c$, sera n^2.
- Para el caso del algoritmo de ordenación de inserción presenta un tiempo de ejecución $\Theta(n^2)$.
- Un algoritmo es mejor que otro al comparar su tiempo de ejecución en el peor caso.

Tasa de crecimiento Θ

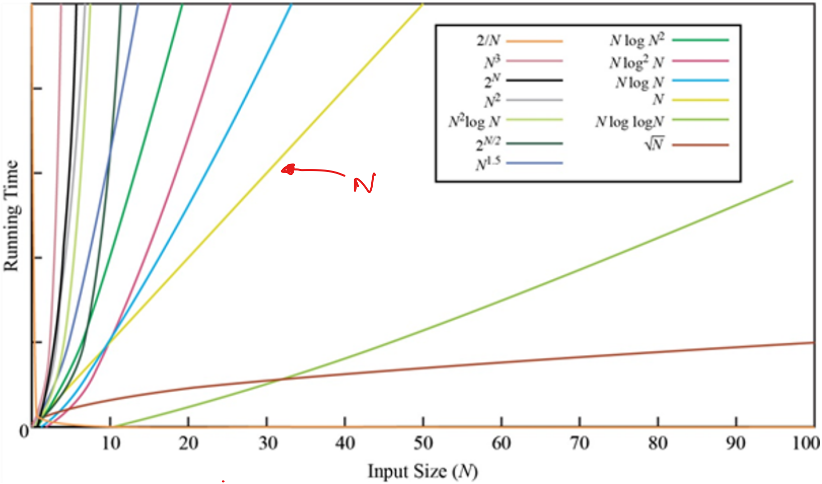
- Se prioriza el término de mayor orden de la fórmula, sin incluir su coeficiente.
- Se asume que los términos de menor orden y constantes son insignificantes. Por ejemplo: $an^2 + bn + c$, sera n^2.
- Para el caso del algoritmo de ordenación de inserción presenta un tiempo de ejecución $\Theta(n^2)$.
- Un algoritmo es mejor que otro al comparar su tiempo de ejecución en el peor caso.

Tasa de crecimiento Θ

- Se prioriza el término de mayor orden de la fórmula, sin incluir su coeficiente.
- Se asume que los términos de menor orden y constantes son insignificantes. Por ejemplo: $an^2 + bn + c$, sera n^2.
- Para el caso del algoritmo de ordenación de inserción presenta un tiempo de ejecución $\Theta(n^2)$.
- Un algoritmo es mejor que otro al comparar su tiempo de ejecución en el peor caso.

Tasa de crecimiento Θ

N^2



N

- Existen un amplio rango de técnicas de algoritmos.
- Ordenación por inserción emplea un enfoque incremental:

Estrategia Divide y Conquista

- Algoritmo de estructura recursivo y toma menos tiempo de ejecución que ordenación por inserción.
- Involucra tres pasos:
  - **Divide:** el problema en un número de subproblemas, pequeñas instancias del problema.
  - **Conquista:** los subproblemas resolviendolos recursivamente. Al ser muy pequeño, se resuelve fácilmente.
  - **Combina:** Los resultados de los subproblemas en un solución al problema original.

- **Divide:** Divide la secuencia de n-elementos para ser ordenado en subsequencia de $n/2$ elementos cada una.
- **Conquista:** Ordenamos las dos subsecuencias recursivamente usando merge sort
- **Combina:** Mezclamos las dos subsecuencias ordenada para producir la respuesta ordenada
- La recursión **termina** cuando la longitud de la subsecuencia es 1, caso que no puede realizar ordenamiento.

- **Divide:** Divide la secuencia de n-elementos para ser ordenado en subsequencia de $n/2$ elementos cada una.
- **Conquista:** Ordenamos las dos subsecuencias recursivamente usando merge sort
- **Combina:** Mezclamos las dos subsecuencias ordenada para producir la respuesta ordenada
- La recursión **termina** cuando la longitud de la subsecuencia es 1, caso que no puede realizar ordenamiento.

Merge-sort

- **Divide:** Divide la secuencia de n-elementos para ser ordenado en subsequencia de $n/2$ elementos cada una.
- **Conquista:** Ordenamos las dos subsecuencias recursivamente usando merge sort
- **Combina:** Mezclamos las dos subsecuencias ordenada para producir la respuesta ordenada
- La recursión **termina** cuando la longitud de la subsecuencia es 1, caso que no puede realizar ordenamiento.

Merge-sort

- **Divide:** Divide la secuencia de n-elementos para ser ordenado en subsequencia de $n/2$ elementos cada una.
- **Conquista:** Ordenamos las dos subsecuencias recursivamente usando merge sort
- **Combina:** Mezclamos las dos subsecuencias ordenada para producir la respuesta ordenada
- La recursión **termina** cuando la longitud de la subsecuencia es 1, caso que no puede realizar ordenamiento.

Time to play!

https://youtu.be/Pr2Jf83_kG0

```
Function Merge(A, p,q,r):
    n1 = q − p + 1
    n2 = r − q
    Crear L[1..n1+] y R[1..n2 + 1]
    for i = 1 to n1 do
        L[i] = A[p + i − 1]
    end
    for j = 1 to n2 do
        R[j] = A[q + j]
    end
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        end
    end
```

- A es un arreglo; p, q y r son índices dentro del arreglo $p \leq q < r$
- El procedimiento asume que los subarreglos $A[p..q]$ y $A[q+1..r]$ estan ordenados; y los mezcla en un solo arreglo ordenado $A[p..r]$
- La mezcla toma un tiempo $\Theta(n)$, $n = r - p +$
- Se emplea un **centinela** para simplificar el código.

Función Merge


```
Function Merge(A, p,q,r):
    n1 = q − p + 1
    n2 = r − q
    Crear L[1..n1+] y R[1..n2 + 1]
    for i = 1 to n1 do
        L[i] = A[p + i − 1]
    end
    for j = 1 to n2 do
        R[j] = A[q + j]
    end
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        end
    end
```

- A es un arreglo; p, q y r son índices dentro del arreglo $p \leq q < r$
- El procedimiento asume que los subarreglos $A[p..q]$ y $A[q+1..r]$ estan ordenados; y los mezcla en un solo arreglo ordenado $A[p..r]$
- La mezcla toma un tiempo $\Theta(n)$, $n = r - p +$
- Se emplea un **centinela** para simplificar el código.

Función Merge


```
Function Merge(A, p,q,r):
    n1 = q − p + 1
    n2 = r − q
    Crear L[1..n1+] y R[1..n2 + 1]
    for i = 1 to n1 do
        L[i] = A[p + i − 1]
    end
    for j = 1 to n2 do
        R[j] = A[q + j]
    end
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        end
    end
```

- A es un arreglo; p, q y r son índices dentro del arreglo $p \leq q < r$
- El procedimiento asume que los subarreglos $A[p..q]$ y $A[q+1..r]$ estan ordenados; y los mezcla en un solo arreglo ordenado $A[p..r]$
- La mezcla toma un tiempo $\Theta(n)$, $n = r - p +$
- Se emplea un **centinela** para simplificar el código.

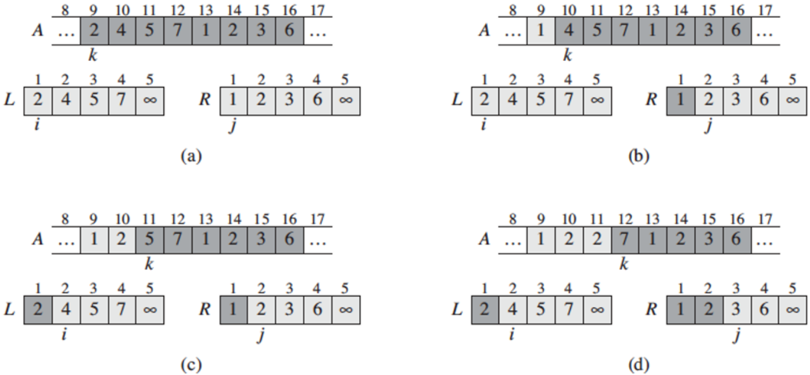
Función Merge

Function `Merge`(A, p, q, r):

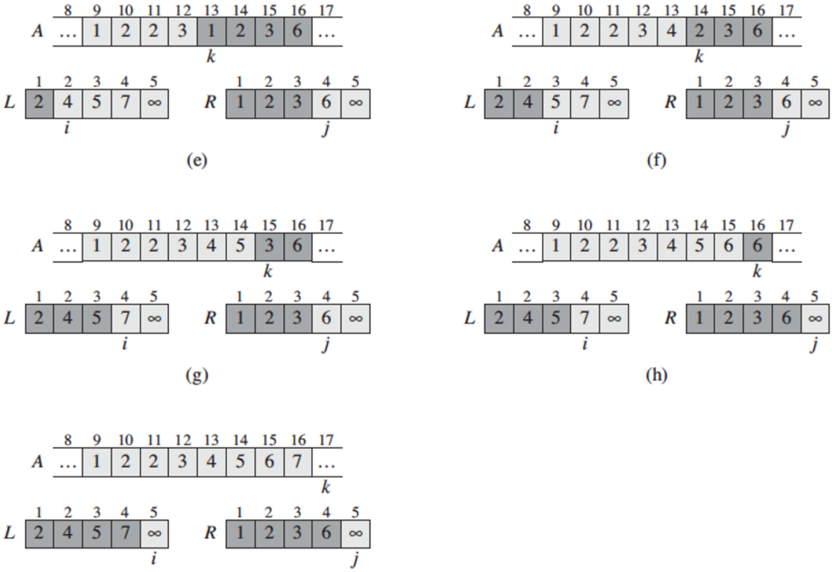
- $n_1 = q - p + 1$
- $n_2 = r - q$
- Crear $L[1..n_1+]$ y $R[1..n_2 + 1]$
- **for** $i = 1$ **to** n_1 **do**
  - $L[i] = A[p + i - 1]$
- **end**
- **for** $j = 1$ **to** n_2 **do**
  - $R[j] = A[q + j]$
- **end**
- $L[n_1 + 1] = \infty$
- $R[n_2 + 1] = \infty$
- $i = 1$
- $j = 1$
- **for** $k = p$ **to** r **do**
  - **if** $L[i] \leq R[j]$ **then**
    - $A[k] = L[i]$
    - $i = i + 1$
  - **else**
    - $A[k] = R[j]$
    - $j = j + 1$
  - **end**
- **end**

- A es un arreglo; p, q y r son índices dentro del arreglo $p \leq q < r$
- El procedimiento asume que los subarreglos $A[p..q]$ y $A[q + 1..r]$ estan ordenados; y los mezcla en un solo arreglo ordenado $A[p..r]$
- La mezcla toma un tiempo $\Theta(n)$, $n = r - p+$
- Se emplea un **centinela** para simplificar el código.

Ejecución de Merge



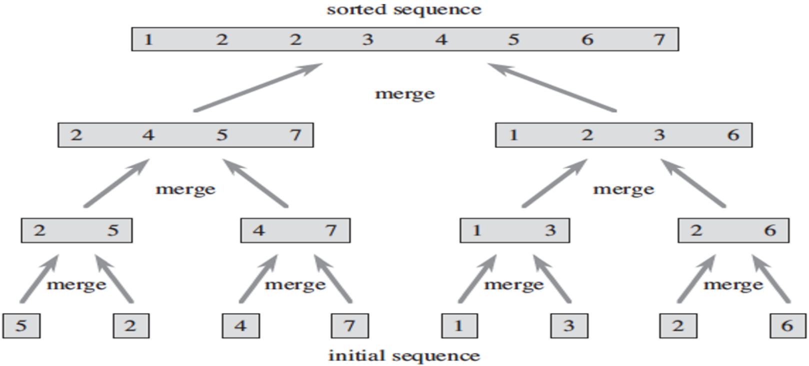
Ejecución de Merge



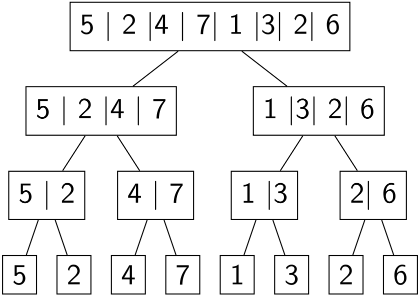
Procedimiento Merge Sort

Time to play!

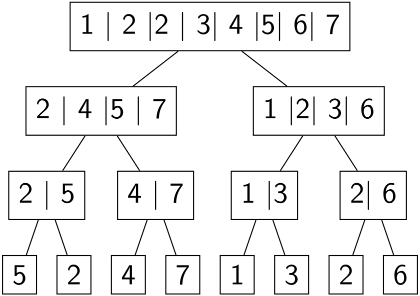
Ejecución de Merge



Merge sort: divide



Merge sort: conquista



Función Merge


```
Function MERGE-SORT(A, p,r):
    if p < r then
        q = (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
  - $A[p..q]$ con $n/2$ elementos.
  - $A[q+1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

Función Merge

Function `MERGE-SORT`(A, p, r):

- **if** $p < r$ **then**
    - $q = (p + r)/2$
    - MERGE-SORT(A, p, q)
    - MERGE-SORT$(A, q + 1, r)$
    - MERGE(A, p, q, r)

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
    - $A[p..q]$ con $n/2$ elementos.
    - $A[q + 1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

Función Merge

```
Function MERGE-SORT(A, p,r):
    if p < r then
        q = (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
  - $A[p..q]$ con $n/2$ elementos.
  - $A[q+1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

Función Merge

```
Function MERGE-SORT(A, p,r):
    if p < r then
        q = (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
  - $A[p..q]$ con $n/2$ elementos.
  - $A[q+1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

Función Merge

Function `MERGE-SORT`$(A,\ p,r)$**:**

```
if p < r then
    q = (p + r)/2
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
  - $A[p..q]$ con $n/2$ elementos.
  - $A[q+1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

Función Merge

```
Function MERGE-SORT(A, p,r):
    if p < r then
        q = (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

- Merge es una subrtuina de MERGE-SORT(A, p, r)
- Se ejecuta si $p \geq r$, por lo menos un elemento y esta ordenado
- Caso contrario se divide A en dos subarreglos:
  - $A[p..q]$ con $n/2$ elementos.
  - $A[q+1..r]$ con $n/2$ elementos.
- Para ordenar $A =< A[1], .., A[n] >$, la llamada del procedimiento será MERGE-SORT$(A, 1, length(A) = n)$

- Una ecuación de recurrencia describe el $T(n)$ del algoritmo recursivo.
- Se basa en los tres pasos del paradigma:
  - Si $n \leq c$ el tiempo de solución es constante $\Theta(1)$
  - Suponemos que dividimos el problema en a subproblemas, cada uno de tamaño $1/b$ del problema original. Esto toma $aT(n/b)$. $D(n)$ tiempo para dividir el problema en subproblemas y $C(n)$ tiempo para combinar las soluciones:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq c. \\ aT(n/b) + D(n) + C(n), & \text{otro caso.} \end{cases} \tag{1}$$

Análisis: Merge sort

Se asume que:

- Tamaño de arreglo n es potencia de 2.
- Cada división es exactamente $n/2$

$T(n)$ es el peor caso:

- Ordenar un elemento toma $\Theta(1)$
- Si $n > 1$ el tiempo de ejecución es descrito:
  - **Divide:** cálculo de la posición toma tiempo constante $D(n) = \Theta(1)$
  - **Conquista:** resuelve recursivamente dos subproblemas de tamaño $n/2$. Entonces $2T(n/2)$ el tiempo de correr.
  - **Combina:** Merge toma un tiempo $\Theta(n)$, $C(n) = \Theta(n)$

Análisis: Merge sort

Al reemplazar términos en la ecuación de recurrencia obtenemos

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ 2T(n/2) + \Theta(n), & \text{if } n > 1. \end{cases} \tag{2}$$

Equivale a

$$T(n) = \begin{cases} c & \text{if } n = 1. \\ 2T(n/2) + cn, & \text{if } n > 1. \end{cases} \tag{3}$$

Análisis: Árbol de recurrencia

$T(n)$

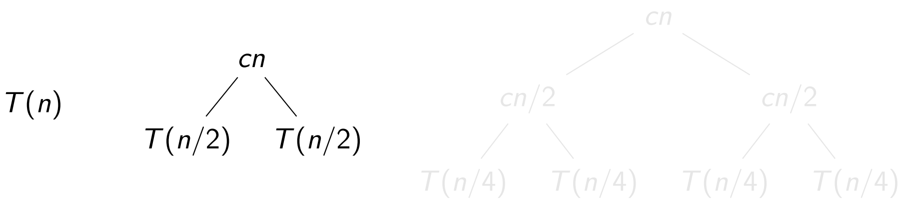
cn

$T(n/2) \quad T(n/2)$

cn

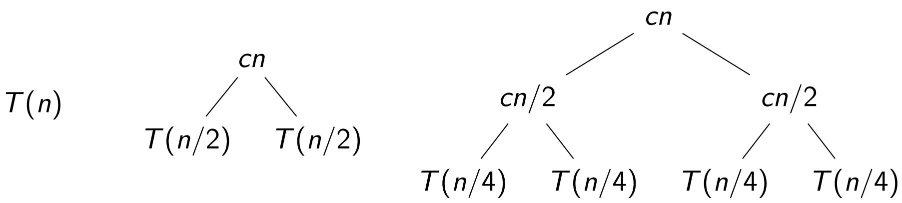
$cn/2 \quad cn/2$

$T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4)$

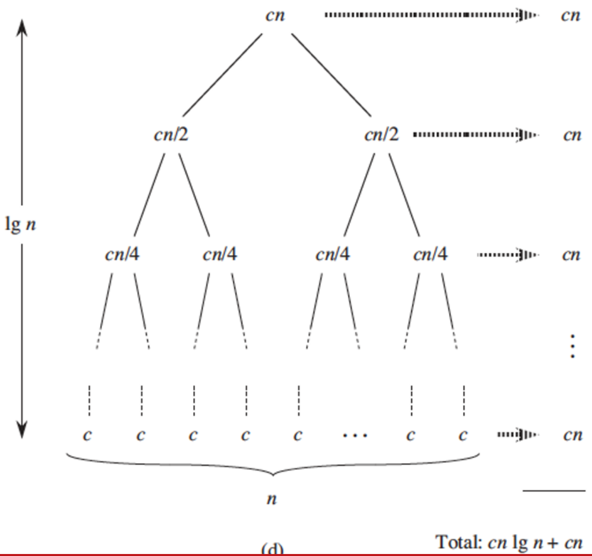


$T(n)$
cn
$T(n/2)$ $T(n/2)$
cn
$cn/2$ $cn/2$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

Análisis: Árbol de recurrencia



Análisis: Árbol de recurrencia



- Se emplea para describir el tiempo recorrido de un algoritmo, es definido en términos del dominio de la función.
- $T(n)$ describe la función del peor tiempo de ejecución.
- Se presentas tres tipos de análisis:
    - **Peor caso:**
        - $T(n) =$ máximo tiempo para cualquier entrada de tamaño n
    - **Caso Promedio:**
        - $T(n) =$ tiempo esperado sobre todas las entradas de tamaño n
    - **Mejor caso:**
        - Se aprovecha de pocos casos donde el algoritmo tiene su mejor ejecución.

Se define $\Theta(g(n)) = \{fn(n):$ existen constantes positivos c_1, c_2 y n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

Ejemplo:
$\frac{n^2}{2} - 3n = \Theta(n^2)$

Notación Asintótica Θ

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

Dividimos por n^2 a los campos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Por la derecha $c_2 \leq \frac{1}{2}$ y $n \geq 1$, en la izquierda $n \geq 7, c_1 \leq \frac{1}{14}$.
Entonces escogamos $n_0 = 7, c_1 = \frac{1}{14}$ y $c_2 = \frac{1}{2}$

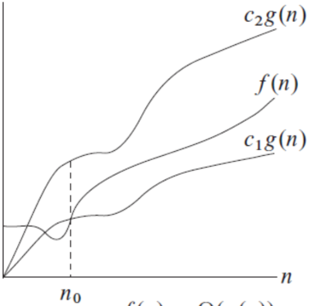
Notación Asintótica Θ

Se define $\Theta(g(n)) = \{fn(n) :$ existen constantes positivos c_1, c_2 y n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

Ejemplo:
$\frac{n^2}{2} - 2n = \Theta(n^2)$ Para las constantes $c_1, c2$ y n_0

Notación Asintótica Θ

- $g(n)$ es un límite asintótico ajustado para $f(n)$
- $f(n) \epsilon \Theta(g(n))$ si encuentra $c_1 g(n)$ y $c_2 g(n)$
- Cada $f(n) = \Theta(g(n))$ tiene que ser asintóticamente positiva, $f(n)$ positiva para un n grande



$f(n) = \Theta(g(n))$

Notación Asintótica Θ

$\frac{1}{3}n^2 - 2n = \Theta(n^2)$

Es necesario encontrar la existencia de c_1, c_2 y n_0:

$c_1 n^2 \leq \frac{1}{3}n^2 - 2n \leq c_2 n^2$

Para todo $n \geq n_0$

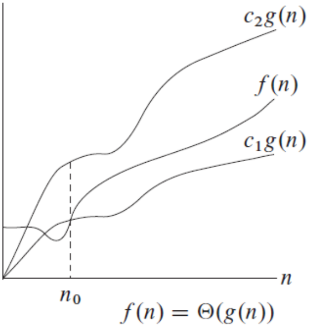
Lado izquierdo $c_1 = \frac{1}{21}$ para $n \geq n_0$

Lado derecho $c_2 = \frac{1}{3}$ para $n \geq 1$

Dividiendo por n^2

$c_1 \leq \frac{1}{3} - \frac{2}{n} \leq c_2$

$c_1 = \frac{1}{21}, c_2 = \frac{1}{3}, n_0 = 7$

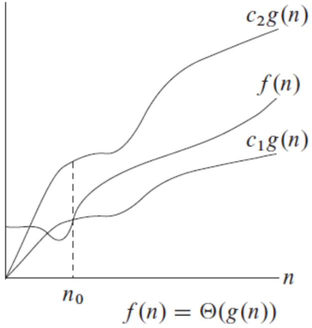


Notación Asintótica Θ

Si $f(n) = an^2 + bn + c$ con $a > 0$ luego $f(n) = \Theta(n^2)$. En general para cualquier:

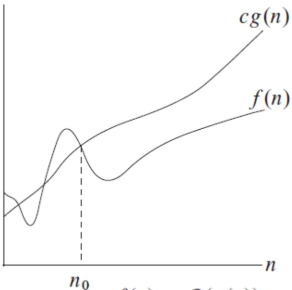
$p(n) = \sum_{i=0}^{d} a_i n^i$ con $a_d > 0$ entonces

$p(n) = \Theta(n^d)$



Notación Asintótica O

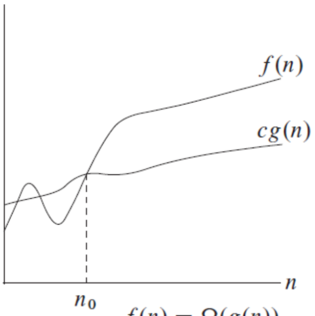
- $g(n)$ es un límite superior de $f(n)$
- $O(g(n))$ es el conjunto de funciones:
- Cada $O(g(n)) = f(n)$: existe una constante positiva c y n_0 tal que $0 \leq f(n) \leq cg(n)$
- Notar que $f(n) = \Theta(g(n))$ implica que $f(n) = O(g(n))$. Ejemplos para $O(n^2)$
  - $n^2 + n$
  - $n^2 - n$
  - $1000n^2 + 1000n$
  - $n^{1,999}$
  - $n/1000$



$f(n) = O(g(n))$

Notación Asintótica Ω

- $g(n)$ es un límite inferior de $f(n)$
- $\Omega(g(n))$ es el conjunto de funciones:
- Cada $\Omega(g(n)) = f(n)$: existe una constante positiva c y n_0 tal que $0 \leq cg(n) \leq f(n)$
- Notar que $f(n) = \Theta(g(n))$ implica que $f(n) = \Omega(g(n))$. Ejemplos para $\Omega(n^2)$
    - $n^2 + n$
    - $n^2 - n$
    - $1000n^2 + 1000n$
    - $n^{1,999}$
    - $n/1000$



$f(n) = \Omega(g(n))$

Recurrrencia

- Es una ecuación o desigualdad que describe función en términos de sus valores más pequeños.
- La recurrencia de merge sort

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ 2T(\frac{n}{2}) + \Theta(n), & n > 1. \end{cases} \tag{4}$$

- Al resolver se convierte en $T(n) = \Theta(nlgn)$
- **Objetivo:** es aplicar técnicas que permitan delimitar en notación Θ a la ecuación de recurrencia.

Método del Árbol de Recursión

- Convierte la recurrencia en un árbol de nodos en un árbol.
- Cada nodo representa el costo de un determinado nivel de recursión.

Método Master

- Provee limites a la recurrencia de forma
  - $T(n) = aT(\frac{n}{b}) + f(n)$
  - Donde $a \geq 1, b > 1$ y $f(n)$ es una función dada

Método del Árbol de Recursión

- Convierte la recurrencia en un árbol de nodos en un árbol.
- Cada nodo representa el costo de un determinado nivel de recursión.

Método Master

- Provee limites a la recurrencia de forma
  - $T(n) = aT(\frac{n}{b}) + f(n)$
  - Donde $a \geq 1, b > 1$ y $f(n)$ es una función dada

Tecnicismos

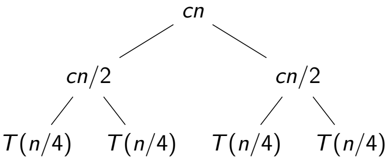
- Normalmente se omiten ciertos aspectos técnicos al resolver recurrencias.
- Ej: Asumir valores enteros como argumentos de las funciones

$$Merge_sort = T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & n > 1. \end{cases} \tag{5}$$

- Condiciones de límite también se omiten, no afectan el orden de crecimiento
- **Importante:** Establecer cuando los detalles importan en el cálculo

Árbol de Recursión

- **Árbol de recurrencia**
  - Cada nodo representa el costo de un subproblema
  - Suma de costos de cada nodo por nivel $\rightarrow$ costo de nivel
  - Suma de costos de niveles *rightarrow* costo del árbol
- **Utilidad:** recurrencia describe el tiempo de ejecución del algoritmo divide y conquista.
- Genera buenas propuestas de solución, y se puede tolerar cierto grado de omisión, ya que se verificarán por el método de substitución.
- Si es cuidadoso al aplicar este método, el árbol de recursión se convertirá en una prueba directa de la solución.



Método del Árbol de Recursión

Método del Árbol de Recursión

Ejemplo: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

- Proveer una buena solución aplicada árbol de recursión.
- Solución:
  - Omitiendo la aproximación, $T(n) = 3T(n/4) + cn^2, c > 0$
  - Asumir que n es una potencia de 4

Método del Árbol de Recursión

$T(n)$

$$cn^2$$
$$T(n/4) \quad T(n/4) \quad T(n/4)$$

$$cn^2$$
$$c(n/4)^2 \qquad c(n/4)^2 \qquad c(n/4)^2$$
$$T(n/16) \; T(n/16) \; T(n/16) \; T(n/16) \; T(n/16) \; T(n/16) \; T(n/16) \; T(n/16) \; T(n/16)$$

Método del Árbol de Recursión

$$T(n)$$

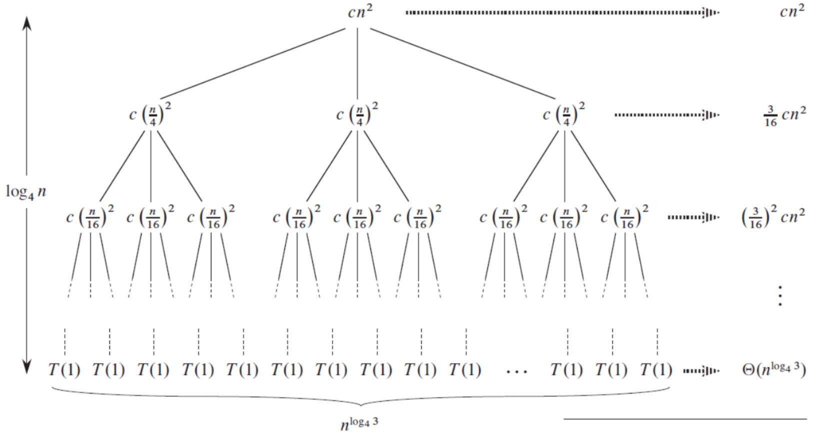
$$cn^2$$

$$T(n/4) \quad T(n/4) \quad T(n/4)$$

$$cn^2$$

$$c(n/4)^2 \qquad c(n/4)^2 \qquad c(n/4)^2$$

$$T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16) \quad T(n/16)$$



cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\log_4 n$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

(d)

Total: $O(n^2)$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

Método del Árbol de Recursión

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

- Tamaño de subproblemas disminuye mientras nos alejamos de la raíz.
- ¿Qué tan lejos de la raíz se alcanza la condición límite?
  - El tamaño del subproblema para un modo a profundidad i es $n/4$.
  - El subproblema alcanza el límite cuando $i = log_4 n$
- El árbol tendrá $log_4 n$ niveles $(0..log_4 n)$
- El árbol tendrá 3^i nodos en el nivel i
- Cada nodo a profundidad $i(0..log_4 N - 1)$ tendrá un costo de $c(n/4^i)^2$

- El costo total de los modos a profundidad $i(0..log_4 n - 1)$ será $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- A profundidad $log_4 n$, se tiene $3^{log_4 n} = n^{log_4 3}$ nodos, con costo $T(1)$, que totaliza $\Theta(log_4 3)$
- La sumatoria total de costos del árbol de recursión es

$$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + .. + (\frac{3}{16})^{log_4 n - 1} cn^2 + \Theta(n^{log_4 3})$$

$$= \sum_{i=0}^{log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{log_4 3})$$

$$= \frac{(3/16) log_4 n - 1}{1 - (3/16)} cn^2 + \Theta(n^{log_4 3})$$

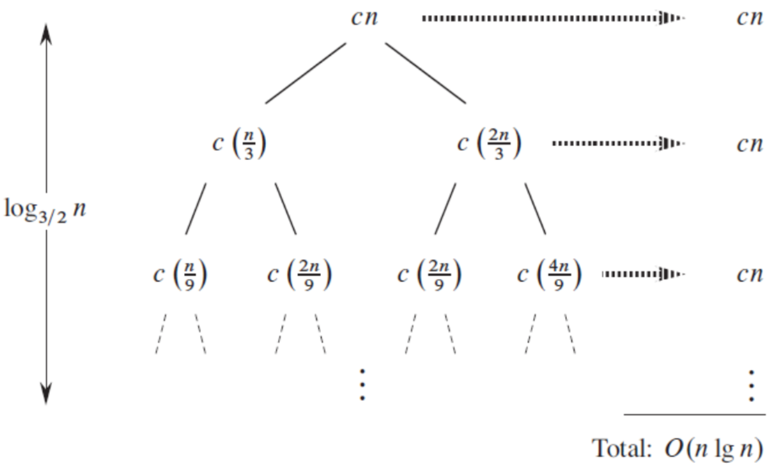
Usando una serie geométrica decreciente infinita como límite superior, se obtiene

$$
\begin{aligned}
T(n) = \quad & cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + .. + (\frac{3}{16})^{log_4 n-1} cn^2 + \Theta(n^{log_4 3}) \\
& < \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{log_4 3}) \\
& = \frac{1}{\frac{3}{16}} cn^2 + \Theta(n^{log_4 3}) \\
& = \frac{16}{13} cn^2 + \Theta(n^{log_4 3}) \\
& = O(n^2)
\end{aligned}
$$

Método del Árbol de Recursión

Probar $T(n) = T(n/3) + T(2n/3) + cn$

Probar $T(n) = T(n/3) + T(2n/3) + cn$



Ejercicios: Árbol de Recursión

Desarrollar

- $T(n) = 2T(n/2) + \Theta(n^4)$
- $T(n) = T(3n/5) + 2T(n/5) + \Theta(n)$
- $T(n) = 2T(n/2) + \Theta(lgn)$

- Soluciona recurrencias de la forma:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- $a \geq 1, b > 1$ constantes positivas.
- $f(n)$ función asintóticamente positiva.
- La recurrencia anterior divide el problema en a subproblemas de tamaño n/b los cuales los resuelve recursivamente en un tiempo $T(n/b)$
- **Combinar y dividir se describe mediante:** $f(n) = C(n) + D(n)$. Ejemplo en Merge-sort $f(n) = \Theta(n)$

Teorema:

- Sea $a \geq 1$ y $b > 1$ constantes, sea $f(n)$ una función, y $T(n)$ se encuentre definido para los enteros no negativos mediante la recurrencia.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- Donde n/b puede ser interpretado como $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces $T(n)$ puede ser acotado asintóticamente como sigue:

- Si $f(n) = O(n^{\log_b} a - \epsilon)$ para alguna constante $\epsilon > 0$ entonces $T(n) = \Theta(n^{\log_b a})$
- Si $f(n) = \Theta(n^{\log_b a})$ entonces $T(n) = \Theta(n^{\log_b a} lgn)$
- Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante $\epsilon > 0$, y si $af(\frac{n}{b}) \leq cf(n)$ para alguna constante $c < 1$ y todos los n suficientes grandes entonces $T(n) = \Theta(f(n))$

Idea:

Comparar $f(n)$ con $n^{\log_b a}$ intuitivamente la solución de la recurrencia está determinada por la función mas grande.

Caso 1

- $f(n)$ debe ser polinómicamente mas pequeña (no solo más pequeña), es decir asintóticamente más pequeña por un factor de n^{ϵ} para $\epsilon > 0$.
- Para el caso 1, $n^{log_b a}$ es mas grande, entonces la solución es $\Theta(n^{log_b a})$

Ejemplo

- $$T(n) = 9T(\frac{n}{3}) + n$$
  $a = 9, b = 3, f(n) = n$,luego
  $n^{log_b a} = n^{log_3 9-\epsilon}$, donde $\epsilon = 1$
- Se puede notar lo siguiente:
  $$f(n) = O(n^{log_3 9-\epsilon}),\ \text{donde } \epsilon = 1$$
- Aplicando el teorema master para el caso I concluimos que la solución $T(n) = \Theta(n^2)$

Caso 2

- Para el caso 2, las dos funciones son del mismo tamaño, multiplicando por lgn, se tiene $\Theta(f(n)lgn)$

Ejemplo

- $$T(n) = T(\frac{2n}{3}) + 1$$
  $a = 1, b = \frac{3}{2}, f(n) = 1$,luego
  $n^{log_b a} = n^{log_{3/2} 1} = \Theta(1)$
- Se puede notar lo siguiente:
  $$f(n) = \Theta(n^{log_{3/2} 1}) = \Theta(1)$$
- Aplicando el teorema master para el caso 2 concluimos que la solución $T(n) = \Theta(lgn)$

Caso 3

- $f(n)$ debe ser **polinómicamente dominante** (no solo más grande) satisfacer la condición de regularidad $af(\frac{n}{b}) \leq cf(n)$.
- Para el caso 3, $f(n)$ es mas grande, entonces la solución es $\Theta(f(n))$

Ejemplo

- $$T(n) = 3T(\frac{n}{4}) + n$$
  $$a = 3, b = 4, f(n) = n \quad \text{,luego}$$
  $$n^{log_b a} = n^{log_4 3} = O(n^{0{,}793})$$
- Se puede notar lo siguiente: $f(n) = O(n^{log_4 3 + \epsilon})$, donde $\epsilon = 0{,}2$
- Aplicando el teorema master para el caso 3 concluimos que la solución $T(n) = \Theta(n)$, si y solo si la condición de regularidad se mantiene: para n, grande $af(\frac{n}{b}) = 3(\frac{n}{4}) \leq (\frac{3}{4})n = cf(n)$, para $c = \frac{3}{4}$

Desarrollar los siguientes items

- $T(n) = 2T(\frac{n}{2}) + \Theta(n^2)$
- $T(n) = T(\frac{n}{2}) + 2^n$
- $T(n) = 2^n T(\frac{n}{2}) + n^n$
- $T(n) = 0{,}5T(\frac{2n}{3}) + \Theta(n^3)$
- $T(n) = 2T(\frac{4n}{7}) + T(\frac{n}{7}) + \Theta(n)$
- $T(n) = \sqrt{2}T(\frac{n}{3}) + \Theta(lgn)$
- $T(n) = 6T(\frac{n}{3}) + n^2 lgn$
- $T(n) = 5T(\frac{n}{5}) + \sqrt{n}$
- $T(n) = 0{,}25T(\frac{n}{4}) + \frac{1}{n}$
- $T(n) = 64T(\frac{n}{8}) - n^2 lgn$

Referencias

T.Cormen.

Introduction to Algorithms. Third Edition.

The MIT Press, 2009, chapter 2