ANÁLISIS DE ALGORITMOS

(RECURSIVOS)

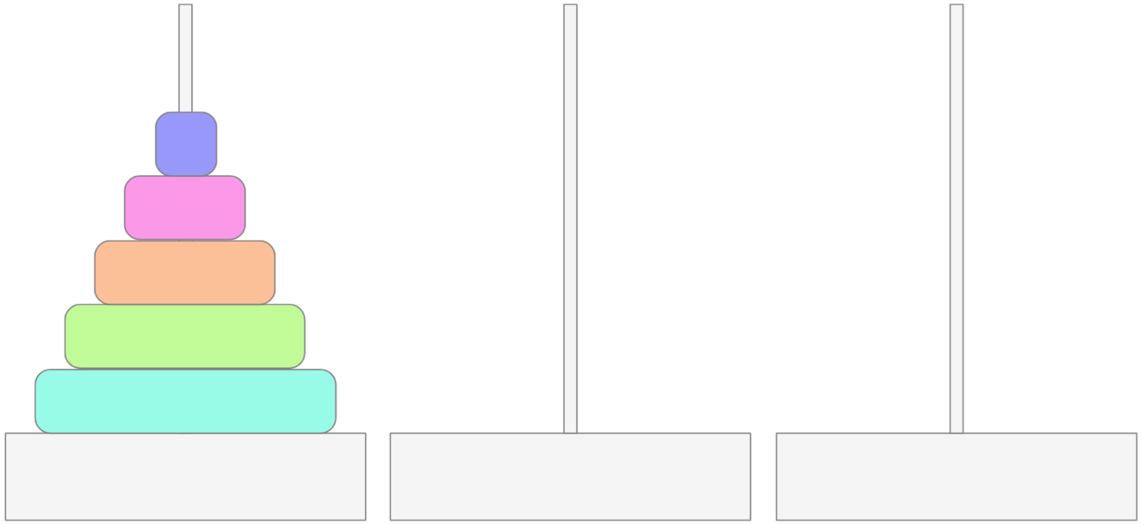
Algoritmos y Programación
Javier Miranda

Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

29 de noviembre de 2023

¿ Cómo analizamos algoritmos recursivos ?

```
def TowerOfHanoi(n, src, dest):
    if n==1:
        MoveDisk (1, src, dest)
    else:
        TowerOfHanoi(n-1, src, 6-src-dest)
        MoveDisk (1, src, dest)
        TowerOfHanoi(n-1, 6-src-dest, dest)
```



http://towersofhanoi.info/Animate.aspx

Paso 1: Obtenemos la recurrencia

`def TowerOfHanoi(n, src, dest):`	t_n
`    if n==1:`	
`        MoveDisk (1, src, dest)`	1
`    else:`	
`        TowerOfHanoi(n-1, src, 6-src-dest)`	t_{n-1}
`        MoveDisk (1, src, dest)`	1
`        TowerOfHanoi(n-1, 6-src-dest, dest)`	t_{n-1}

1. Establecemos el tamaño del ejemplar dependiendo de los parámetros del algoritmo
2. Añadimos el coste de las llamadas recursivas
3. Añadimos el coste de ejecución de la parte iterativa
4. Identificamos y añadimos los casos base

¿ Recurrencia ?

$$\begin{aligned} t_n &= 2t_{n-1} + 1 \quad n > 1 \\ t_1 &= 1 \end{aligned}$$

Paso 2: Resolvemos la recurrencia

$$t_n = 2t_{n-1} + 1 \quad n > 1$$
$$t_1 = 1$$

1. Método de sustitución (*forward / backward*)

2. Aplicando el teorema maestro

   Sólo para recurrencias de divide y vencerás (o reduce y vencerás)

3. Utilizando una herramienta de resolución de recurrencias (*solver de recurrencias*)

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

$$t_n = 2t_{n-1} + 1 \quad n > 1$$
$$t_1 = 1$$

2.1 Método de sustitución (forward)

$$\left.\begin{array}{l} t_1 = 1 \\ t_2 = 2^*t_1 + 1 = 2^*1 + 1 = 3 \\ t_3 = 2^*t_2 + 1 = 2^*3 + 1 = 7 \\ t_4 = 2^*t_3 + 1 = 2^*7 + 1 = 15 \\ \ldots \end{array}\right\} \quad t_n = 2^n - 1$$

Se conoce también como Guess & Test

$\in$ **O (2^n)**

2. Solución mediante el teoréma maestro

$$T(n) \leq \underset{\text{Número de llamadas recursivas}}{a} \cdot T\left(\frac{n}{\underset{\text{Número de divisiones}}{b}}\right) + \underset{\text{Coste de la fase de combinación}}{O(n^d)}.$$

Número de llamadas recursivas

Número de divisiones

Coste de la fase de combinación

$$T(n) = \begin{cases} O(n^d \log n) & \textit{if } a = b^d \\ O(n^d) & \textit{if } a < b^d \\ O(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

Caso 1

Caso 2

Caso 3

Sólo para recurrencias de problemas resueltos mediante divide y vencerás (o reduce y vencerás)

3. Solución mediante un solver

$$t_n = 2t_{n-1} + 1 \quad n > 1$$
$$t_1 = 1$$

WolframAlpha computational intelligence.

t(1)=1, t(n)=2*t(n-1) + 1

Browse Examples

Input:

$t(1) = 1 \quad | \quad t(n) = 2\,t(n-1) + 1$

Alternate form:

$\{t(1) = 1,\ t(n) = 5\,t(n-1) + 1\}$

Recurrence equation solution:

$\boxed{t(n) = 2^n - 1}$

https://www.wolframalpha.com/examples/mathematics/discrete-mathematics/recurrences/

3. Solución mediante un solver

PURRS: The Parma University's Recurrence Relation Solver

x(n) = 2*x(n-1) + 1

Initial conditions: x(1)=1

Solve or approximate! Clear ☐ Verify the solution

This is computed on an AMD processor running GNU/Linux.

Exact solution for x(n) = 1+2*x(-1+n) for the initial conditions x(1) = 1

x(n) = -1+2^n
for each n >= 1

http://www.cs.unipr.it/purrs/

Ejemplos

Algoritmos Iterativos

- Búsqueda binaria
- Mochila 0/1 *(Greedy)*

→ • Mochila 0/1 *(Programación Dinámica: Tabulation)*

- Algoritmos Recursivos
- Búsqueda binaria
- Merge Sort
- Quick Sort

Mochila 0/1 (fuerza bruta)

$$t(n,w) = \begin{cases} \boxed{t(n-1, w) \qquad : W_n > w} \\ \max(t(n-1,w), t(n-1,w-W_n) + B_n) \\ \boxed{0 \qquad : n <= 0} \end{cases}$$

$$t(n) = 1 + t(n-1)$$
$$t(0) = 1$$

t(n)=1+t(n-1); t(0)=1

Recurrence equation solution

$$t(n) = n + 1$$

Mochila 0/1 (fuerza bruta)

$$t(n,w) \begin{cases} t(n-1, w) & : W_n > w \\ \boxed{\begin{array}{ll} \max(t(n-1,w), t(n-1,w-W_n) + B_n) & \\ 0 & : n <= 0 \end{array}} \end{cases}$$

$$t(n) = 1 + t(n-1)$$
$$t(0) = 1$$

t(n)=1+t(n-1); t(0)=1

Recurrence equation solution

$$t(n) = n + 1$$

$$t(n) = 1 + 2*t(n-1)$$
$$t(0) = 1$$

t(n)=1+2*t(n-1); t(0)=1

Recurrence equation solution

$$t(n) = 2^{n+1} - 1$$

$O(2^n)$

Programación Dinámica: Tabulation Mochila 0/1

1. Definir N
2. Casos de estudio
3. Reglas de análisis

 # Fase 1 del algoritmo: Rellenar la tabla

```
for w = 0 to W                  <------ O(w)
    V[0,w] = 0

for i = 1 to n                  <------ O(n)
    V[i,0] = 0

for i = 1 to n                  <------ O(n * w)
    for w = 0 to W
        if w_i <= w             <------ Operación crítica
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]        O(1)
            else
                V[i,w] = V[i-1,w]
        else
            V[i,w] = V[i-1,w]                       O(1)
```

$$O\ (\boldsymbol{n}\ *\boldsymbol{w})$$

n\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Programación Dinámica: Tabulation Mochila 0/1


```
# Fase 1 del algoritmo: Rellenar la tabla

for w = 0 to W
    V[0,w] = 0

for i = 1 to n
    V[i,0] = 0

for i = 1 to n
    for w = 0 to W
        if w_i <= w
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]
            else
                V[i,w] = V[i-1,w]
        else
            V[i,w] = V[i-1,w]
```

$O\ (n\ *w)$

```
# Fase 2 del algoritmo: Utilizando el contenido
# de la tabla identificar los items elegidos

i=n , k=W

while i >0:
    if V[i,k] ≠ V[i–1,k] then
            // El i^th elemento está en la mochila
        i = i–1, k = k-w_i
    else
            // El i^th elemento no está en la mochila
        i = i–1
```

$O\ (n)$

n\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$\boxed{O\ (n\ *w)}$

Programación Dinámica: Tabulation
Mochila 0/1

$$\boxed{O\ (\boldsymbol{n} * \boldsymbol{w})}$$

- A primera vista, con programación dinámica, tenemos una solución polinomial
- Sin embargo, en todas las publicaciones y en internet se dice que es un algoritmo **pseudo-polinomial**

¿ Por qué ?

Porque es un algoritmo que con ejemplares del mismo tamaño (el mismo número de ítems N) es muy sensible a variaciones en W.

Ejemplos

<u>Algoritmos Iterativos</u>

→ • Búsqueda binaria

• Mochila 0/1 *(Greedy)*

• Mochila 0/1 *(Programación Dinámica: Tabulation)*

• <u>Algoritmos Recursivos</u>

→ • Búsqueda binaria

• Merge Sort

• Quick Sort

Búsqueda Binaria *(iterativa)*

```
def binary_search(arr, x):
    low  = 0
    high = len(arr) - 1
    mid  = 0

    while low <= high:
        mid = (high + low) // 2   # Integer Floor Division

        if arr[mid] < x:
            low = mid + 1

        elif arr[mid] > x:
            high = mid - 1

        else:
            return mid

    return -1
```

O(log n)

Operación crítica

https://en.wikipedia.org/wiki/Floor_and_ceiling_functions

$\in$ O (log n)

Búsqueda Binaria *(recursiva)*

```
binSearch (a, left, right, value) {
    if (right < left) return Not_Found
    mid = (left + right) / 2

    if a[mid] > value
        return binSearch (a, left, mid-1, value)
    elsif a[mid] < value
        return binSearch (a, mid+1, right, value)
    else
        return mid
```

a = número de llamadas recursivas (en cada division)
b = factor de reducción de la entrada (fase de división)
d = coste de la fase de combinación

a = 1; b = 2; d=0 → $a = b^d$

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

$$T(n) = \begin{cases} O(n^d \log n) & \textit{if } a = b^d \\ O(n^d) & \textit{if } a < b^d \\ O(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

Caso 1 → $\boxed{O\ (\log n)}$

Merge Sort

```
MergeSort (a, left, right) {
    if (left < right) {
       mid = divide (a, left, right)
       MergeSort (a, left, mid)
       MergeSort (a, mid+1, right)
       merge(a, left, mid+1, right)
    }
}
```

Pseudocode for Merge:

C = output [length = n]
A = 1st sorted array [n/2]
B = 2nd sorted array [n/2]
i = 1
j = 1

```
for k = 1 to n
        if A(i) < B(j)
                C(k) = A(i)
                i++
        else [B(j) < A(i)]
                C(k) = B(j)
                j++
end
```

(ignores end cases)

$O(n)$

Merge Sort

```
MergeSort (a, left, right) {
    if (left < right) {
        mid = divide (a, left, right)
        MergeSort (a, left, mid)
        MergeSort (a, mid+1, right)
        merge(a, left, mid+1, right)
    }
}
```

O(n)

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

Merge Sort

```
MergeSort (a, left, right) {
    if (left < right) {
        mid = divide (a, left, right)
        MergeSort (a, left, mid)
        MergeSort (a, mid+1, right)
        merge(a, left, mid+1, right)
    }
}
```

$\leftarrow$ *O(n)*

a = número de llamadas recursivas (en cada division)
b = factor de reducción de la entrada (fase de división)
d = coste de la fase de combinación

a = 2; b = 2 ; d=1 $\rightarrow$ $a = b^d$

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

$$T(n) = \begin{cases} O(n^d \log n) & \mathit{if}\ a = b^d \\ O(n^d) & \mathit{if}\ a < b^d \\ O(n^{\log_b a}) & \mathit{if}\ a > b^d \end{cases}$$

Caso 1 $\dashrightarrow$ $\boxed{O\ (n \log n)}$

QuickSort: Mejor caso

Analizamos dos casos: mejor caso y peor caso

`QuickSort (a, left, right)`	t_n
`  if (left < right)`	1
`    pivot = Partition (a, left, right)`	n
`    Quicksort (a, left, pivot-1)`	$t_{n/2}$
`    Quicksort (a, pivot+1, right)`	$t_{n/2}$

<u>Mejor Caso</u>

$$t_n = n + 2t_{n/2}$$

$$t_0 = t_1 = 1$$

QuickSort: Mejor caso

$$t_n = n + 2t_{n/2}$$
$$t_0 = t_1 = 1$$

Utilizando el solver de recurrencias: http://www.cs.unipr.it/purrs/

x(n) = 2*x(n/2) + n

Initial conditions: x(1) = 1

Solve or approximate! | Clear | ☑ Verify the solution

This is computed on an AMD processor running GNU/Linux.

x(n) >= 2-3/2*n+n*log(n)*log(2)^(-1)

x(n) <= n+n*log(n)*log(2)^(-1)

for each n >= 1

$\in$ **O(n log n)**

QuickSort: Peor caso

```
QuickSort (a, left, right)                              t_n
  if (left < right)                                     1
     pivot = Partition (a, left, right)                 n
     Quicksort (a, left, pivot-1)
     Quicksort (a, pivot+1, right)                      t_{n-1}
```

<u>Peor Caso</u>

$t_n = n + t_{n-1}$

$t_0 = t_1 = 1$

QuickSort: Peor caso

$$t_n = n + t_{n-1}$$
$$t_0 = t_1 = 1$$

Utilizando el solver de recurrencias: http://www.cs.unipr.it/purrs/

x(n) = x(n-1) + n

Initial conditions: x(0) = 1; x(1) = 1

Solve or approximate! Clear Verify the solution

This is computed on an AMD processor running GNU/Linux.

x(n) = 1/2*n^2+1/2*n
for each n >= 1

$\in O(n^2)$

Ejemplos

Algoritmos Iterativos

- Búsqueda binaria
- → Mochila 0/1 *(Greedy)*
- Mochila 0/1 *(Programación Dinámica: Tabulation)*

- Algoritmos Recursivos
- Búsqueda binaria
- Merge Sort
- Quick Sort

Knapsack 0/1 (Greedy)

<u>Algoritmo Base</u>

Paso 1: Ordenar los N elementos

Knapsack 0/1 (Greedy)

<u>Algoritmo Base</u>

Paso 1: Ordenar los N elementos **O (?)**

Paso 2: Recorrer los elementos ordenados hasta llenar al máximo posible la mochila **O (n)**

<u>Método de Ordenación</u>

- Burbuja ……… $O(n^2)$
- Selección …… $O(n^2)$
- Inserción ……. $O(n^2)$
- Merge Sort ………. $O(n*\log n)$
- Quick Sort ……….. $O(n*\log n)$, Peor caso = $O(n^2)$

Knapsack 0/1 (Greedy)

<u>Algoritmo Base</u>

Paso 1: Ordenar los N elementos — **O (??)**

Paso 2: Recorrer los elementos ordenados hasta llenar al máximo posible la mochila — **O (n)**

Python

Operation	Average Case
Sort	O(n log n)

https://wiki.python.org/moin/TimeComplexity

Knapsack 0/1 (Greedy)

Algoritmo Base

Paso 1: Ordenar los N elementos **O (n*log n)**

Paso 2: Recorrer los elementos ordenados hasta llenar al máximo posible la mochila **O (n)**

Python

Operation	Average Case
Sort	O(n log n)

O (n*log n)

https://wiki.python.org/moin/TimeComplexity

Resumen (1/2)

- Notación Big-O
  - Cota superior del coste del algoritmo

- Análisis asintótico de código iterativo:
  - Paso 1: Definir N
  - Paso 2: Aplicar reglas de análisis asintótico

  - Variantes:
    - Expandida: considerando todas las estructuras de control de flujo que componen el algoritmo
    - Abreviada: considerando sólamente el coste de la operación crítica

Resumen (2/2)

- Análisis asintótico de código recursivo:
  - Paso 1: Obtener la recurrencia
  - Paso 2: Resolver la recurrencia
    - Sustitución
    - Método maestro
    - Utilizando una herramienta de resolución de recurrencias