Algoritmos y Estructuras de Datos I

Carreras:

Lic. En Informática

Programador Universitario

Facultad de Ciencias Exactas y Tecnología

2020

Técnicas de diseño de Algoritmos

- Fuerza bruta
- Recursión
- Dividir para Conquistar (Divide & Conquer)
- Programación dinámica
- Técnica ambiciosa (Greedy)
- Vuelta atrás (Backtracking)
- Ramificación y poda (Branch and Bound)
- Algoritmos Probabilistas.

Recursión

Un objeto es recursivo cuando se define en función de si mismo.

El concepto de recursividad se usa mucho en la matemática pero también en la computación.

Una función se dice recursiva si se invoca a si misma.

Ej. La definición recursiva de la función factorial:

n!: ***entero ≥0 → entero ≥1***

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Recursión

El uso de la recursión permite escribir algoritmos **sencillos y concisos**.

```
FUNCION Fac(n) : entero ≥0 → entero ≥1
        SI n=0 ENTONCES
                retorna (1)
        SINO
                retorna ( n * Fac (n-1) )
 FIN
```

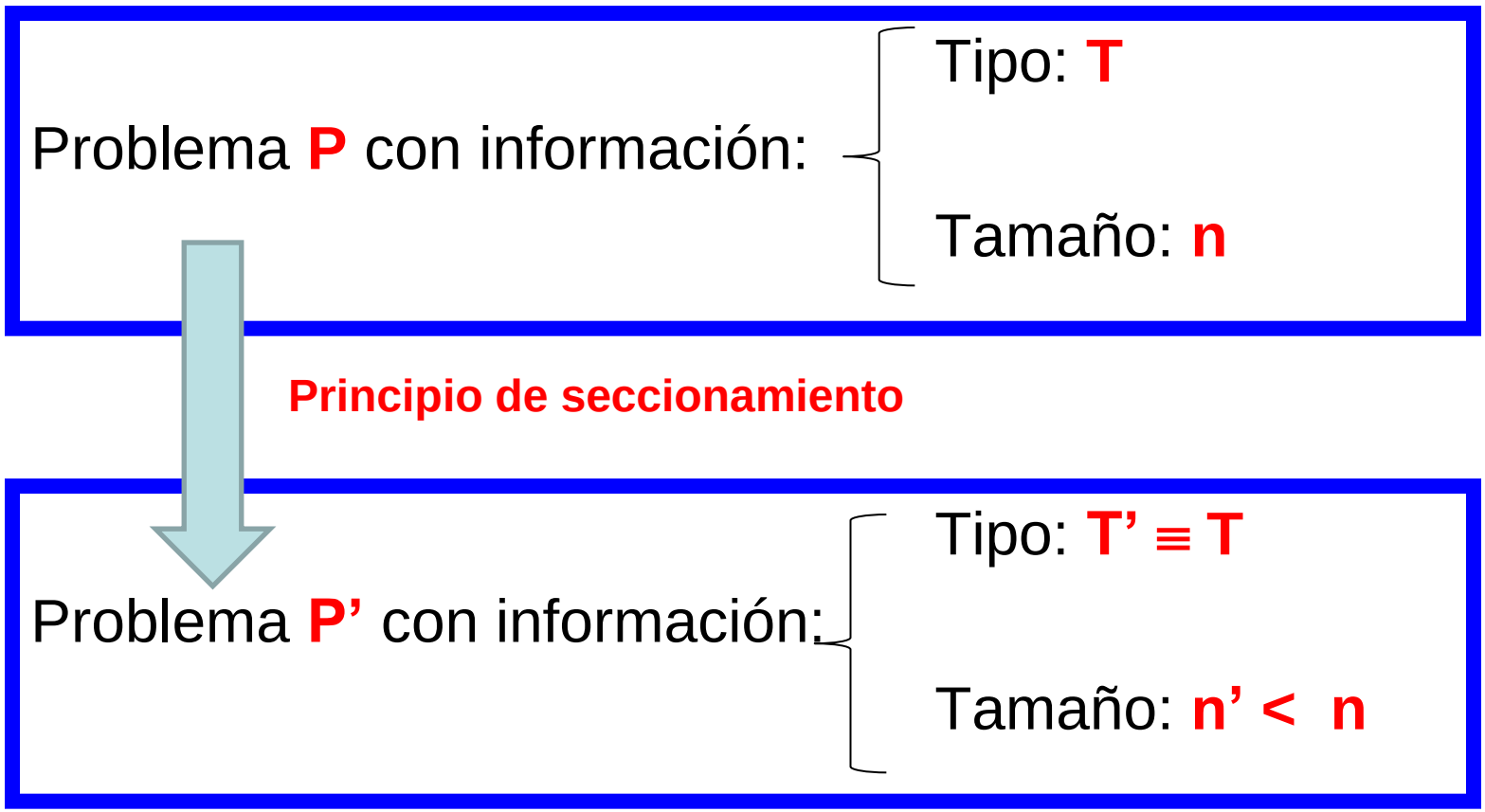
Recursión

Principios generales del análisis recurrente

Dado un problema **P** que trata información de un cierto tipo y de un tamaño, se quiere reducir el problema **P**, expresando el tratamiento de la información de ***igual tipo*** **pero de** ***menor tamaño*****.**

Es importante hacer una análisis del problema para encontrar un medio de seccionar esta información en subinformaciones de igual tipo.

Recursión


Problema P con información:
Tipo: T
Tamaño: n
Principio de seccionamiento
Problema P' con información:
Tipo: T' ≡ T
Tamaño: n' < n

Recursión

Este tipo de análisis requiere gran precisión y se puede guiar con las siguientes etapas:

- definición de las informaciones tratadas
- principio de seccionamiento
- expresión de los cálculos
- comprobación de las soluciones obtenidas

La búsqueda de las relaciones de recurrencia es la parte central del trabajo de análisis. Puede hacerse de manera deductiva o de manera directa inductiva.

Recursión-Ejemplo x^n

Ejemplo: calcular x^n, cuando n es un entero positivo o nulo y x es un numero real.

pot(x,n) : real x entero ≥ 0 → real

1) Algoritmo usando *iteracion*:

Se define como el producto de n copias de x.

x*x*x….x (n veces)

2) Algoritmo usando ***recursión***:

Se define x^n en función de x^{n-1}

$x^n = x * x^{n-1}$ si $n \geq 1$

$x^0 = 1$

Ejemplo- Algoritmo 1

Algoritmo de potencia iterativo:

```
FUNCION pot1(x,n) : real x entero ≥0 → real
        SI n=0 ENTONCES
                p ←1
        SINO
                p ← x
                PARA i=1,n-1 HACER
                        p ← p*x
        retorna (p)
 FIN
```

- La complejidad de pot1 es **O(n)**

Ejemplo- Algoritmo 2

Algoritmo de potencia usando: simplificación + recursión

```
FUNCION pot2(x,n) : real x entero ≥0 → real
        SI n=0 ENTONCES
                retorna (1)
        SINO
                retorna ( x * pot2(x,n-1) )
FIN
```

- La complejidad de pot2 es **O(n)**
- **Como se calcula la complejidad en las funciones recursivas?**

Recursión

Si se tienen funciones recursivas, se debe asociar una función de tiempo desconocida T(n).

- Se plantea una ***recurrencia para T(n)***, donde n mide el tamaño de la entrada al proceso.
- Se generaliza la ecuación para T(n) en términos de T(k) para distintos valores de k.
- Se particulariza para algun valor apropiado de k llegando al caso base.

Recursión-Ejemplo pot2

FUNCION pot2(x,n) : real x entero ≥0 → real

```
        SI n=0 ENTONCES
                retorna (1)
        SINO                    // n ≥ 1
                retorna ( x * pot2(x,n-1) )
FIN
```

Sea T(n) el tiempo para calcular pot2, su forma recurrente:

$$T(n) = \begin{cases} d & \text{si } n = 0 \\ c + T(n-1) & \text{si } n \geq 1 \end{cases}$$

c,d son constantes

Recursión-Ejemplo pot2

Sea T(n) el tiempo para calcular pot2

$T(n) = d$ si $n = 0$

$T(n) = c + T(n-1)$ si $n \geq 1$ c,d son constantes

desarrollando la recurrencia:

si $n \geq 1$ $T(n)=c+T(n-1)$

si $n-1 \geq 1$, $n \geq 2$ $T(n-1)=c+T(n-2)$, $T(n)=2*c+T(n-2)$

si $n-2 \geq 1$, $n \geq 3$ $T(n-2)=c+T(n-3)$, $T(n)=3*c+T(n-3)$

…generalizando:

$\forall\ n \geq k$ $T(n)=k*c+T(n-k)$

en particular, vale para n=k entonces:

si n=k $T(n)=n*c+T(n-n)=n*c+T(0)= n*c+d$

entonces T(n) es O(n)

Recursión-Ejemplo Fac

```
FUNCION Fac (n) : entero  ≥0 → entero ≥1
        SI n=0 OR n=1  ENTONCES retorna (1)
        SINO retorna ( n * Fac (n-1) )
FIN
```

Sea T(n) el tiempo para calcular Fac(n):

$$T(n) = \begin{cases} d & \text{si } 0 \le n \le 1 \\ c + T(n-1) & \text{si } n > 1 \end{cases}$$

c,d son constantes

Recursión-Ejemplo Fac

Sea T(n) el tiempo para calcular Fac(n)

T(n) = d si $n \leq 1$

T(n) = c + T(n-1) si n > 1 c,d son constantes

desarrollando la recurrencia:

si n>1 T(n)=c+T(n-1)

si n-1>1, n>2 T(n-1)=c+T(n-2), T(n)=2*c+T(n-2)

si n-2>1, n>3 T(n-2)=c+T(n-3), T(n)=3*c+T(n-3)

…generalizando:

$\forall$ n>k T(n)=k*c+T(n-k)

en particular, vale para n=k+1 entonces:

si n=k+1 T(n)=(n-1)*c+T(1)=c*(n-1)+d

entonces T(n) es O(n)

Recursión-Ejemplo Binario

Escribir una función recursiva que muestre el código binario de un entero dado.

Ej. $25_{10} \rightarrow (11001)_2$

25 / 2 = 12 resto (25 / 2) = 1

12 / 2 = 6 resto (12 / 2) = 0

6 / 2 = 3 resto (6 / 2) = 0

3 / 2 = 1 resto (3 / 2) = 1

Recursión-Ejemplo Binario

```
Función Binario (n) :Entero≥1 en base 10
                    → escribe secuencia de dígitos binarios
    Si n = 1 ENTONCES
        Escribir (n)
    SINO                        // es n > 1
        Binario(n/2)
        Escribir (n mod 2)
FIN
```

Relación de recurrencia:

$T(n) = d$ si $n = 1$

$T(n) = c + T(n/2)$ si $n \geq 2$ c,d son constantes

Recursión-Ejemplo Binario

Sea

$T(n) = d$ si $n = 1$

$T(n) = c + T(n/2)$ si $n \geq 2$ c,d son constantes

Si $n \geq 2$ $T(n) = c + T(n/2)$

Si $n/2 \geq 2$, $n \geq 4$ $T(n/2) = c + T(n/4)$ $T(n) = 2*c + T(n/4)$

Si $n/4 \geq 2$, $n \geq 8$ $T(n/4) = c + T(n/8)$ $T(n) = 3*c + T(n/8)$

…generalizando:

$\forall\ n \geq 2^k$ $T(n) = k*c + T(n/2^k)$

Vale para $n=2^k$ $T(n) = c* \log_2 n + T(1) = c * \log_2 n + d$

$k= \log_2 n$

entonces $T(n)$ es $O(\log_2 n)$ para Binario

Recursión-Ejemplo f1

```
FUNCION f1 (n) : entero → entero
    SI (n = 0) ENTONCES
        retorna (0)
    SINO
        retorna ( f1(n-1) + f1(n-1) + n * (n+1) )
FIN
```

Relación de recurrencia:

$T(n) = d$ si $n = 0$

$T(n) = c + 2 * T(n-1)$ si $n > 0$ c,d son constantes

Recursión-Ejemplo f1

Sea

$T(n) = d$ si $n = 0$

$T(n) = c + 2 * T(n-1)$ si $n > 0$ c,d son constantes

Si $n>0$ $T(n) = c + 2 * T(n-1)$

Si $n-1>0$, $n>1$ $T(n) = c + 2 * (c + 2 * T(n-2)) = 3 * c + 4 * T(n-2)$

Si $n-2>0$, $n>2$ $T(n) = 7 * c + 8 * T(n-3)$

Si $n-3>0$, $n>3$ $T(n) = 15 * c + 16 * T(n-4)$

$\forall$ $n>k$ $T(n) = (2^{k+1} - 1) * c + 2^{k+1} * T(n-(k+1))$

Si $n=k+1$ $T(n) = (2^n - 1) * c + 2^n * T(0) = 2^n * (c + d) - c$

entonces $T(n)$ es $O(2^n)$ para la función f1

Recursión-Ejemplo f2

```
FUNCION f2 (n) : entero → entero
    SI (n = 0) ENTONCES
        retorna (0)
    SINO
        retorna ( 2 * f2(n-1) + n * (n+1) )
FIN
```

Relación de recurrencia:

T(n) = d si n = 0

T(n) = c + T(n-1) si n > 0 c,d son constantes

Recursión-Ejemplo f2

Sea

$T(n) = d$ si $n = 0$

$T(n) = c + T(n-1)$ si $n > 0$ c,d son constantes

Si n>0 $T(n) = c + T(n-1)$

Si n>1 $T(n) = c + (c + T(n-2)) = 2 * c + T(n-2)$

Si n>2 $T(n) = 3 * c + T(n-3)$

Si n>3 $T(n) = 4 * c + T(n-4)$

$\forall$ n>k $T(n) = (k+1) * c + (n-(k+1))$

Para n=k+1 $T(n) = n * c + T(0) = n * c + d$

entonces T(n) es O(n) para la función f2

Recursión-Ejemplo Multiplicar

Algoritmo de multiplicación por sumas sucesivas:

```
FUNCION M(A,B) :entero≥0 x entero>0 → entero≥0
  si B=1 entonces
          retorna (A)
  sino
          retorna ( A+ M(A,B-1) )
 FIN
```

Relación de recurrencia:

T(A,B) = d si B = 1

T(A,B) = c + T(A,B-1) si B > 1, c,d son constantes

Recursión-Ejemplo Multiplicar

Sea

$T(A,B) = d$ si $B = 1$

$T(A,B) = c + T(A,B-1)$ si $B > 1$, c,d son constantes

Si $B>1$ $T(A,B) = c + T(A,B-1)$

Si $B-1>1$, $B>2$ $T(A,B-1) = c + T(A,B-2)$, $T(A,B) = 2*c + T(A,B-2)$

Si $B-2>1$, $B>3$ $T(A,B-2) = c + T(A,B-3)$, $T(A,B) = 3*c + T(A,B-3)$

$\forall$ $B>k$ $T(A,B) = k*c + T(A,B-k)$

Para $B=k+1$ $T(A,B) = (B-1)*c + T(A,1) = B*c-c+d$

entonces $T(A,B)$ es $O(B)$ para la función M

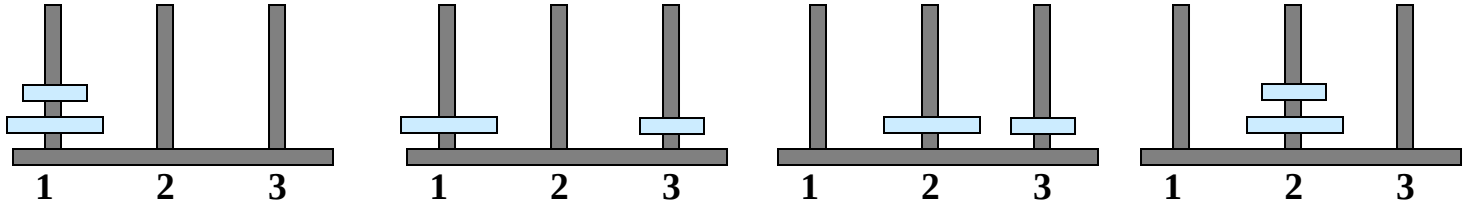
Recursión-Ejemplo Torres Hanoi

Torres de Hanoi (Édouard Lucas-1883)

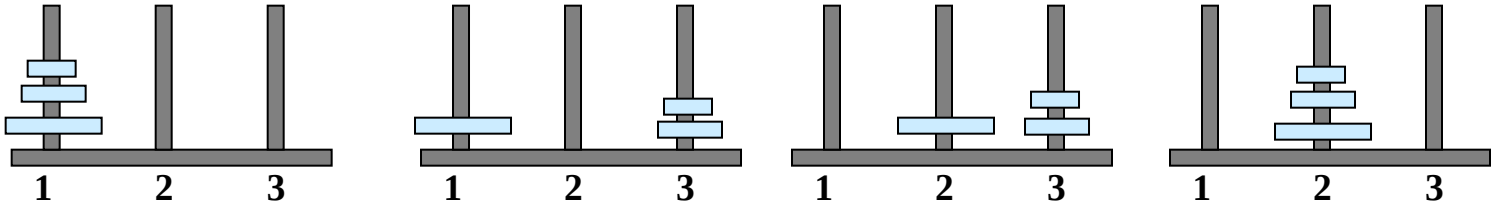
Recursión-Ejemplo Torres Hanoi

Problema de las Torres de Hanoi.

2 anillos: 3 movimientos



3 anillos: 7 movimientos



Recursión-Ejemplo Torres Hanoi

```
FUNCION Hanoi ( n , i , j )
    // mueve los n anillos de la torre i a j //
    SI  n > 0 ENTONCES
        Hanoi ( n-1 , i , 6-i-j )
        Escribir ( se mueve un disco de i → j )
        Hanoi ( n-1 , 6-i-j , j )
    FIN
```

Cuántos movimientos hace este algoritmo para n discos?

Recursión-Ejemplo Torres Hanoi

Sea M(n) el numero de movimiento que se realizan:

$M(n) = 0$ si $n=0$

$M(n) = 2\ M(n-1) + 1$ si $n>0$

desarrollando se puede obtener que **$M(n) = 2^n - 1$**

Entonces el algoritmo HANOI tiene un tiempo ϵ **$O(2^n)$.**

Si n=64 discos:

nro de movimientos = $2^{64} - 1 \approx 1.84\ 10^{19}$

si se hace 1 movimiento / seg durante las 24 hs del dia,

1 año = 31536000 segundos

Tiempo Total = **$5.8\ 10^{11}$ años**

Recursión

- Siempre se puede remover la recursión de un algoritmo y transformarlo en iterativo?
- Es una técnica eficiente la recursión ?
- Cuando conviene usarla ?
- Cuando no se debe usar ?

Divide & Conquer

Es una metodología ***top-down*** ya que resuelve los problemas de una manera *descendente* en 2 etapas:

1. ***Divide:*** división en problemas mas chicos que se resuelven en forma independiente

2. ***Conquer:*** la solución del problema original se forma combinando la solución de los problemas más chicos.

- Generalmente se usa la ***recursión*** en las implementaciones de estos algoritmos, aunque puede aplicarse también de manera ***iterativa***.

Divide & Conquer

ALGORITMO : D&C

ENTRADA: x (problema)
SALIDA: y (solución)

P1. **SI** x es "suficientemente chico" **ENTONCES**
aplicar un algoritmo básico para x
SINO
Descomponer x en instancias mas chicas (x_1, x_2, …, x_k)
PARA i=1,k **HACER**
$y_i \leftarrow$ D&C(x_i)
Combinar los (y_1, y_2, …, y_k) para obtener la solucion y.
P2. **FIN**

Divide & Conquer-Ejemplo x^n

Ejemplo: calcular x^n, cuando n es un entero positivo o nulo y x es un numero real.

1) Algoritmo usando iteración: pot1

2) Algoritmo usando recursión: pot2

3) Algoritmo usando: *divide & conquer + recursión + balance*

Se define x^n en función de $x^{n/2}$

$x^0 = 1$

$x^n = x^{n/2} * x^{n/2}$ si $n \geq 2$ es par

$x^n = x * x^{(n-1)/2} * x^{(n-1)/2}$ si $n \geq 1$ es impar

Divide & Conquer-Ejemplo x^n

Algoritmo usando: recursion + divide & conquer + balance.

```
FUNCION pot3(x,n) : real x entero ≥0 → real
        SI n=0 ENTONCES
                RETORNA (1)
        SINO
                SI n es par ENTONCES
                        RETORNA ( pot3(x *x, n/2))
                SINO
                        RETORNA ( x * pot3(x *x, (n-1)/2))
 FIN
```

- La complejidad de pot3 es **$O(\log_2 n)$**

Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado A[1..n] buscar un dado x por el método de Búsqueda Binaria.

La **búsqueda binaria**, también conocida como **búsqueda dicotómica**, o **búsqueda logarítmica**, determina la pertenencia y/o encuentra la posición de un dado valor en un arreglo ordenado.

Durante este proceso va comparando el valor dado con el elemento x en la posición del medio del arreglo,

- si x coincide con el elemento del medio: listo, lo encontró,
- si x es menor que el del medio: continua con la parte izquierda del arreglo,
- si x es mayor que el del medio: continua con la mitad derecha
- Así sigue hasta que lo encuentre o se crucen los índices.

Ejemplo de D&C: **búsqueda binaria iterativa en A ordenado**

```
FUNCION BBinarialTER(buscado,A,n) : entero x arreglo x entero → bool
P1.   inferior ← 1
P2.   superior ← n
P3.   MIENTRAS (inferior ≤ superior) HACER
            medio ← ( inferior + superior ) / 2
            SI buscado = A[medio] ENTONCES
                  Retorna Verdadero
            SINO SI  buscado < A[medio] ENTONCES
                  superior ← medio – 1
            SINO   // buscado > A[medio]
                  inferior ← medio + 1
P4.   Retorna Falso
P5.   FIN
```

Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado A[1..n] buscar un dato buscado por el método de Búsqueda Binaria.

Ejemplo: n=8 buscado=46

A= [5 11 23 35 46 62 79 94]

inferior=1 superior=8 medio=4 A[4]=35**<buscado**

A= [5 11 23 35 **46 62 79 94]**

inferior=5 superior=8 medio=6 A[6]=62**>buscado**

A= [5 11 23 35 **46** 62 79 94**]**

inferior=5 superior=5 medio=5 A[5]=**46=buscado**

→ encontrado, índice=5

Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado A[1..n] buscar un dato buscado por el método de Búsqueda Binaria.

Ejemplo: n=8 buscado=13

A= [5 11 23 35 46 62 79 94]

inferior=1 superior=8 medio=4 A[4]=35**>buscado**

A= [5 11 23 35 46 62 79 94]

inferior=1 superior=3 medio=2 A[2]=11**<buscado**

A= [5 11 **23** 35 46 62 79 94]

inferior=3 superior=3 medio=3 A[3]=23**>buscado**

inferior=3 superior=2 → se cruzan los índices

→ NOT encontrado, índice= -1

Ejemplo de D&C: **búsqueda binaria recursiva en A ordenado**

```
FUNCION BBinariaR(buscado,A,inferior,superior)
                          : entero x arreglo x entero x entero → bool
P1.   medio ← (inferior + superior) / 2
P2.   SI inferior > superior ENTONCES
            Retorna Falso
      SINO SI buscado = A[medio] ENTONCES
            Retorna Verdadero
      SINO SI buscado < A[medio] ENTONCES
            Retorna BBinariaR(buscado,A,inferior,medio-1)
      SINO  // buscado > A[medio]
            Retorna BBinariaR(buscado,A,medio+1,superior)
P3.   FIN
```

Divide & Conquer

Tiempo de ejecución:

$$T(n)= a\ T(n/b) + c\ n^k$$

Teorema: la solución de la ecuación:

$$T(n)= a\ T(n/b) + c\ n^k$$

donde $c \geq 0$ real, $a \geq 1$ real, $b \geq 2$ entero y $k \geq 0$ entero, es:

- $T(n) = O\ (n^{\log_b a}\ )$ si $a > b^k$
- $T(n) = O\ (\ n^k \log_b n)$ si $a = b^k$
- $T(n) = O\ (\ n^k\ )$ si $a < b^k$

Divide & Conquer

Tiempo de ejecución de Búsqueda binaria:

$$T(n) = a\,T(n/b) + c\,n^k$$

$a=1$

$b=2$

$k=0$

→ $a = b^k$

la solución de la ecuación según el teorema es entonces:

- $T(n) = O(n^k \log_b n)$

→ ***T(n) es O ($\log_2 n$)***

Programación Dinámica

- Es una ***técnica ascendente*** ***(bottom up)***
- Comienza con los casos más pequeños.
- Combinando soluciones, va obteniendo soluciones para los casos cada vez mayores, hasta que finalmente llega a la solución del problema original.
- Forma una tabla de resultados de subproblemas para alcanzar la solución del problema.
- No recalcula.

Programación Dinámica

- La mayor aplicación de la Programación Dinámica es en la resolución de ***problemas de optimización***.

- En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se busca es encontrar la solución de valor óptimo (máximo o mínimo).

- La solución de problemas mediante esta técnica se basa en el llamado principio de optimalidad enunciado por *Bellman en 1957* que dice:

"***En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima***".

Programación Dinámica - Ejemplo

Leonardo Fibonacci (1170-1240), llamado Leonardo Pisano, matemático italiano que realizó importantes aportes en los campos matemáticos del álgebra y la teoría de números.

Descubrió la sucesión llamada de Fibonacci **:** 0, 1, 1, 2, 3, 5, 8, 13…

Los términos se pueden definir por la siguiente recurrencia:

$$f_0 = 0 \quad , \quad f_1 = 1 \quad ,$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{para } n \geq 2$$

A cada término de esta sucesión se le denomina número de Fibonacci.

Programación Dinámica - Ejemplo

El algoritmo obtenido directamente de la definición es ***recursivo:***

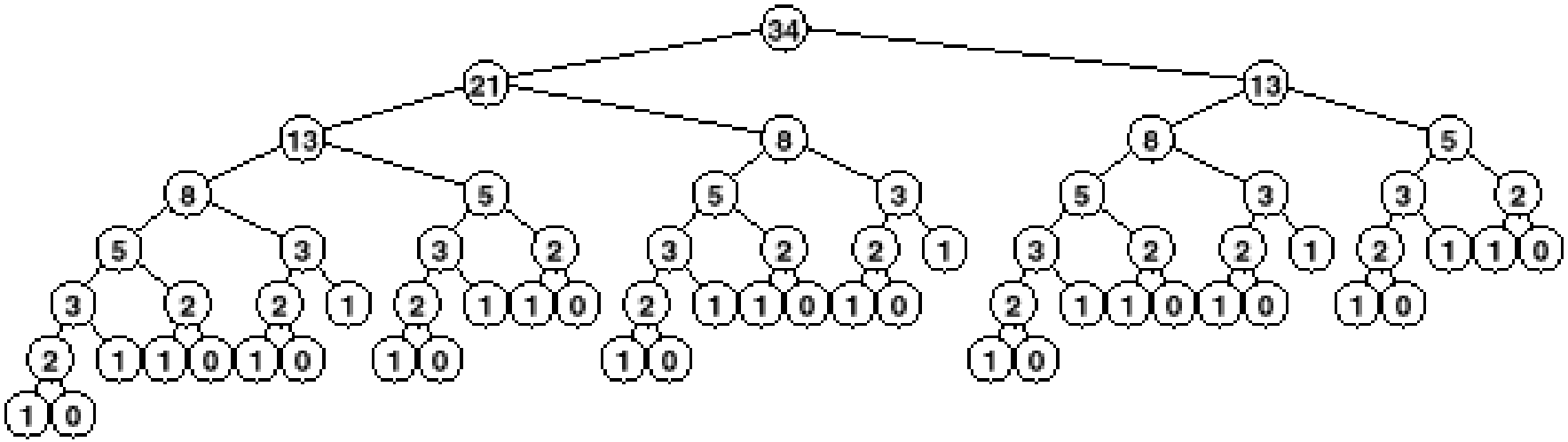
```
FUNCION Fib1 (n) :entero≥0 → entero≥0
    SI  n < 2  ENTONCES
        retorna(n)
    SINO
        retorna Fib1(n-1) + Fib1(n-2)
FIN
```

$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \qquad T(n) \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.618^n)$$

- Fib1 es un algoritmo exponencial en tiempo de ejecución.

Programación Dinámica - Ejemplo

Por ejemplo para Fib1(9)=34 se genera el siguiente árbol de llamadas:



Cabe destacar que se puede plantear otros algoritmos mas eficientes.

Programación Dinámica - Ejemplo

El planteo de ***programación dinámica*** para el calculo directo de la secuencia de Fibonacci hace el almacenamiento en solo 2 variables (i,j).

```
FUNCION Fib2 (n) : entero≥0 → entero≥0
    i ← 1
    j ← 0
    PARA k = 1,n HACER
                j ← i + j;
                i← j - i;
    retorna(j)
    FIN
```

- El algoritmo Fib2 tiene tiempo de orden lineal **O(n).**

Programación Dinámica - Ejemplo

El tercer algoritmo un poco más sofisticado, usa unas cuantas variables auxiliares

```
FUNCION Fib3(n) : entero≥0 → entero≥0
      i ← 1; j ← 0; k ← 0; h ← 1;
      MIENTRAS n>0 HACER
            SI n es impar ENTONCES
                  t ← j*h;
                  j ←i*h+j*k+t;
                  i ←i*k+t;
            t ← h*h;
            h ←2*k*h+t;
            k ←k*k+t;
            n  ← n / 2;
      Retorna(j)
      FIN
```

- El algoritmo Fib3 tiene tiempo de orden **$O(\log_2 n)$**

Programación Dinámica - Ejemplo

Cálculo de **coeficientes binomiales**

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad 0 < k < n \qquad \binom{n}{n} = 1 \qquad \binom{n}{0} = 1$$

```
FUNCION C(n,k) : entero>0 x entero ≥0 → entero>0
    SI k=0 o k=n ENTONCES
        retorna(1)
    SINO
        retorna (C(n-1,k-1) + C(n-1,k))
FIN
```

El algoritmo hace 2*combinaciones(n,k) -2 llamadas recursivas

Programación Dinámica - Ejemplo

Para resolver por programación dinámica se usa una tabla C(0..n,0..k).

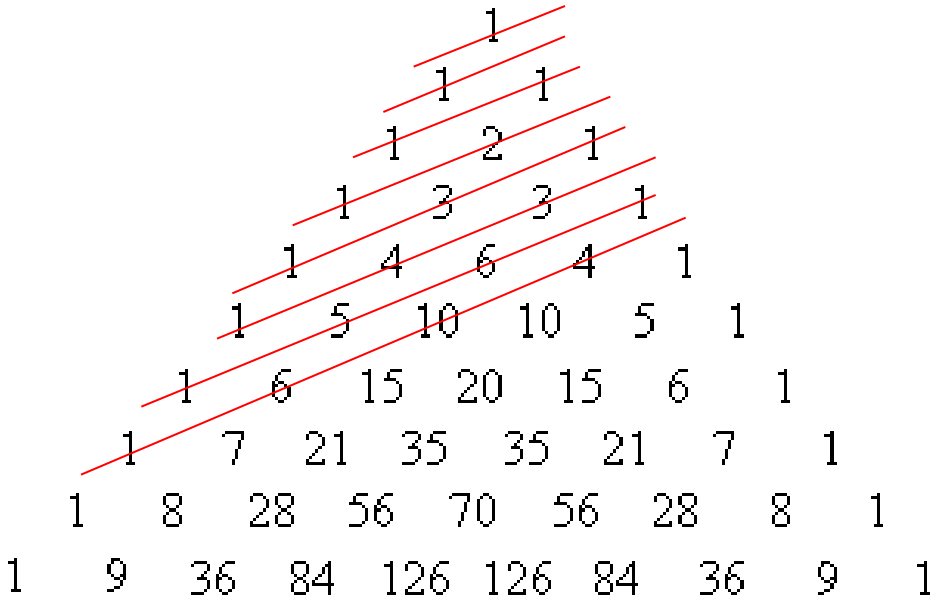
La tabla se llena por fila de izquierda a derecha usando 2 elementos de la fila anterior: C(i,j) = C(i-1,j-1) + C(i-1,j)

	0	**1**	**2**	**3**	**…**	**k-1**	**k**
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
…							
n-1	1					C(n-1,k-1)	C(n-1,k)
n	1						C(n,k) = C(n-1,k-1) + C(n-1,k)

El algoritmo tiene: tiempo $\in$ O(n.k), y usa almacenamiento $\in$ O(n.k)

Programación Dinámica - Ejemplo

Triángulo de Pascal o Triángulo de Tartaglia

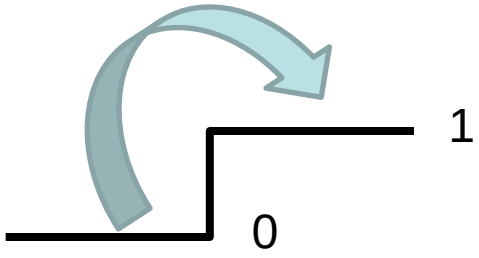


Programación Dinámica - Ejemplo

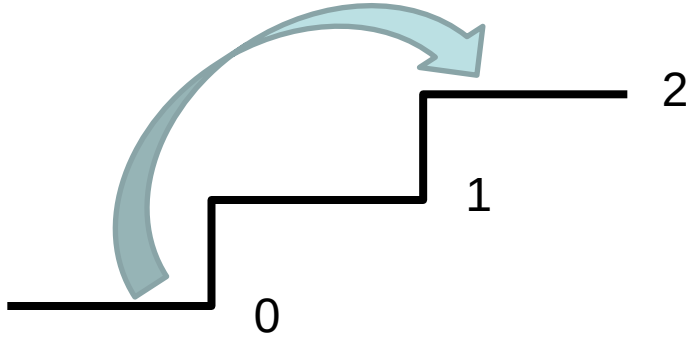
Subir una escalera:

Dos pasos posibles:

1) Al escalón siguiente

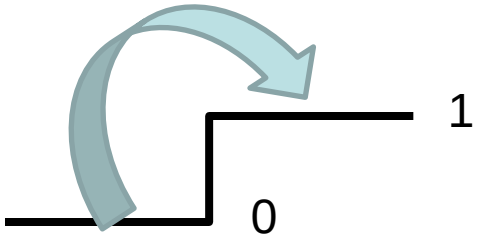


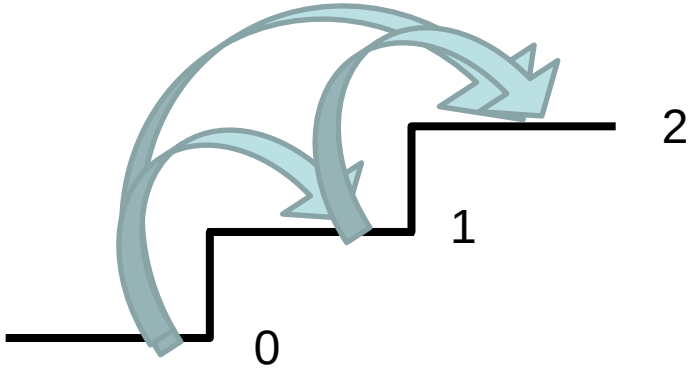
2) Salteando un escalón



Cuántas formas distintas de llegar al escalón **n** partiendo de **0** ?

Programación Dinámica - Ejemplo





Escalon 1: se llega de una sola manera

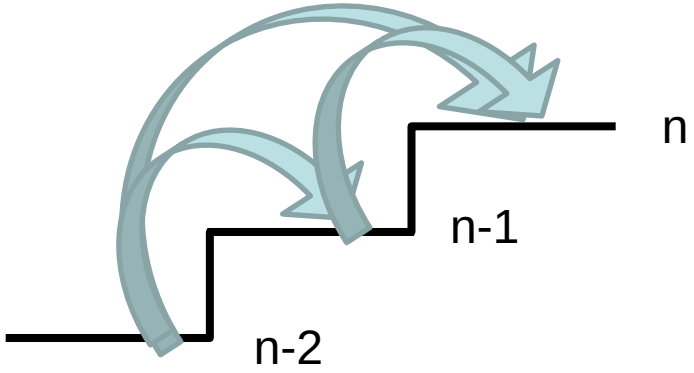
NUM(1) = 1

Escalon 2: se llega de dos maneras distintas.

NUM(2) = 2

Programación Dinámica - Ejemplo

Escalon n: ?



NUM(n) = NUM(n-1) + NUM(n-2) para n>2

Programación Dinámica - Ejemplo

```
FUNCION ESCALERA(n) : entero ≥1 → entero ≥1
  SI n=1 ENTONCES
        retorna (1)
  SINO
     SI n=2 ENTONCES
                retorna 2
      SINO       // n>2
                retorna ESCALERA(n-1) + ESCALERA(n-2)
FIN
```

Será eficiente este algoritmo ?

Como será la implementación con Programación Dinámica ?

Técnica Greedy

El nombre de algoritmo *greedy* se debe a su comportamiento:

- Avanzan siempre hacia "la solución más prometedora".

Los algoritmos greedy son:

- sencillos,
- fáciles de inventar,
- fáciles de implementar,
- cuando funcionan, son eficientes.
- generalmente utilizan la estrategia top-down

- Se usan para resolver *problemas de optimización*.

Técnica Greedy

Se aplica a problemas de optimización que se pueden resolver mediante una ***secuencia de decisiones*.**

Un algoritmo greedy:

- trabaja en una secuencia de pasos
- en cada etapa hace una elección local que se considera una *decisión óptima*,
- luego se resuelve el subproblema que resulta de esta elección,
- finalmente estas soluciones localmente óptimas se integran en una solución global óptima.

Para cada problema de optimización se debe demostrar que la elección óptima en cada paso, lleva a una solución óptima global.

Frecuentemente se preprocesa la Entrada o se usa una una Estructura de Datos adecuada para hacer rápida la elección greedy y así tener un algoritmo más eficiente.

Técnica Greedy

- Dado un problema con n entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema.

- Cada uno de los subconjuntos que cumplan las restricciones se dice que son *soluciones prometedoras.*

- Una solución prometedora que *maximice* o *minimice* una *función objetivo* se denomina *solución óptima*.

- Cuando el algoritmo greedy funciona correctamente, la primera solución que encuentre es siempre óptima.

Técnica Greedy

Elementos que deben estar presentes en el problema:

- Un conjunto de ***candidatos***, que corresponden a las *n* entradas del problema.
- Una ***función de selección*** que en cada momento determina el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos forman ***una solución prometedor***a.
- Una función que comprueba si un subconjunto de estas entradas es ***solución*** al problema, sea óptima o no.
- Una ***función objetivo*** que determina el valor de la solución encontrada. Es la función que se quiere maximizar o minimizar.

Técnica Greedy

En un **algoritmo Greedy** se tiene generalmente:

C: conjunto de candidatos que nos sirven para construir la solución del problema.

S: conjunto de los candidatos ya usados para armar la solución

solución: una función que chequea si un conjunto es solución del problema, ignorando en principio si esta solución es óptima o no.

factible: una función que chequea si un conjunto de candidatos es factible como solución del problema sin considerar si es óptima o no.

selección: una función que indique cual es el candidato mas prometedor que no se eligió todavía. Está relacionada con la función objetivo.

objetivo: una función que es lo que se trata de optimizar. (Esta función no aparece en el algoritmo).

Esquema Algoritmo Greedy

```
FUNCIÓN Greedy (C): conjunto → conjunto
    S ← ∅
    MIENTRAS not solución (S) AND C ≠ ∅ HACER
        x ← selección (C)
        C ← C - {x}
        SI factible ( S U {x} ) ENTONCES
            S ← S U {x}
    FIN MIENTRAS
    RETORNA (S)
FIN
```

Técnica Greedy

En los algoritmos Greedy el proceso no finaliza al diseñar e implementar el algoritmo que resuelve el problema en consideración.

Hay una tarea extra muy importante:

- **SI el algoritmo FUNCIONA BIEN**: la ***demostración formal*** de que el algoritmo encuentra la solución óptima en todos los casos.

- **SI el algoritmo NO FUNCIONA**: la presentación de un ***contraejemplo*** que muestra los casos en donde falla.

Técnica Greedy
Ejemplo: mínimo de monedas

Funcion darvuelto(vuelto): entero → conjunto de monedas

C ← {25, 10, 5, 1} con suficiente cantidad de c/u

S ← ∅

suma ← 0

MIENTRAS suma ≠ vuelto HACER

x ← el elemento de mayor valor en C

C ← C-{x}

SI suma+x ≤ vuelto ENTONCES

S ← S u {x}

suma ← suma+x

FIN MIENTRAS

RETORNA S

Este algoritmo funciona para cualquier valuación ?

Técnica Greedy
Ejemplo: mínimo de monedas

- Se puede demostrar que con los valores de monedas sugeridos (valores de 1, 5, 10 y 25 unidades),

- si hay disponibles de todas las denominaciones de monedas en cantidad suficiente en el conjunto inicial,

- Entonces ***este algoritmo Greedy*** que elige en cada paso la moneda de mayor valor**,** ***siempre encontrará la solución optima*****.**

Técnica Greedy
Ejemplo: mínimo de monedas

Se puede demostrar que con los valores de monedas de los circulantes en Argentina,

el algoritmo Greedy que elige en cada paso la *moneda de mayor valor*,

siempre encontrará la solución optima si es que existe una solución.

Técnica Greedy
Ejemplo: mínimo de monedas

En Europa las monedas son de valor:

{1, 2, 5, 10, 20, 50 } centavos de euro

y de 1€ y 2€.

Funcionará la técnica Greedy planteada?

Se puede demostrar que funciona el algoritmo Greedy

Técnica Greedy
Ejemplo: mínimo de monedas

En el Reino Unido las monedas son de valor:

{1, 2, 5, 10, 20, 25, 50 } peniques (centavos de libra esterlina)

y de 1, 2 y 5 £ (libra esterlina)

Funcionará la técnica Greedy planteada?

Ejemplo:

Para pagar 40 peniques:

Solución greedy usa: 25 + 10 + 5 (3 monedas)

Solución óptima usa: 20 + 20 (2 monedas)

No funciona el algoritmo Greedy

Técnica Greedy
Ejemplo: Problema de Carga

- Un barco se tiene que cargar con contenedores.
- Los contenedores son todos del mismo tamaño, pero de distintos pesos.
- La capacidad de carga del barco esta prefijada.
- Se quiere cargar el barco con el ***máximo número de contenedores.***

Está demostrado que este problema se puede resolver con una técnica greedy obteniendo siempre la solución óptima.

Técnica Greedy
Ejemplo: Problema de Carga

Datos:

n contenedores:

p_i= peso de cada contenedor i, $i=1,2,3,...n$

M= capacidad máxima de carga del barco

Solución: vector X

$x_i = 0$ si el contenedor i no se carga en el barco

$x_i = 1$ si el contenedor i si se carga en el barco

Maximizar la cantidad: $\sum_{i=1}^{n} x_i$

Restricción: $\sum_{i=1}^{n} p_i x_i \leq M$

SOLUCIÓN: Estrategia greedy en peso: en cada paso elegir el contenedor ***de menor peso posible*.**

Técnica Greedy
Ejemplo: Problema de Carga

ALGORITMO: carga
ENTRADA: pesos: vector de nros. reales
capacidad: nro. real
n: entero
SALIDA: x: vector de valores 0 y 1
Leer (n, pesos, capacidad)
Usar un método de ordenación para dar valores a un arreglo t[1..n] de modo que: pesos[t[i]] ≤ pesos[t[i+1]] para i=1..n
Inicializar: i ←1 ; sigue ← true ; x[1..n] ← 0
MIENTRAS (i ≤ n) and sigue HACER
SI pesos[t[i]] ≤ capacidad ENTONCES *// se puede cargar*
x[t[i]] ← 1
capacidad ← capacidad - pesos[t[i]]
i ← i+1
SINO *// capacidad colmada*
sigue ← false

Técnica Greedy
Ejemplo: Problema de la mochila

Este problema es una variante del problema de carga ya presentado.

- Se tiene ***n*** **objetos** y una mochila para llevarlos.
- Cada objeto tiene un ***peso*** y un ***beneficio*** asociado
- La mochila puede cargar un ***peso máximo*** dado.
- El objetivo es llenar la mochila de tal manera que se ***maximice el beneficio de los objetos transportados***, respetando la limitación de la capacidad impuesta.

Técnica Greedy
Ejemplo: Problema de la mochila

Datos: n objetos con sus pesos y beneficios y capacidad de carga:
$i=1,2,3,...n$

p_i= peso de cada objeto i

b_i= beneficio asociado al objeto i

M= capacidad máxima de la mochila

Solución: vector X
$i=1,2,3,...n$

$x_i = 0$ si el objeto i no va en la mochila

$x_i = 1$ si el objeto i se carga en la mochila

Objetivo:

Maximizar la cantidad: $\sum_{i=1}^{n} b_i x_i$

Restricción: $\sum_{i=1}^{n} p_i x_i \leq M$

Tiene solución con la Técnica Greedy?

Técnica Greedy
Ejemplo: Problema de la mochila

PROBLEMA DE LA MOCHILA:

1. ***Greedy al mayor beneficio*.**

Ejemplo: n=3, M=105 , p=[100,10,10], b=[20,15,15]

Solución Greedy : x=[1,0,0] , Beneficio Total = 20

Solución óptima : x=[0,1,1] , Beneficio Total = 30

2. ***Greedy para el menor peso*.**

Ejemplo: n=2, M=25, p=[10,20], b=[5,100]

Solución Greedy en peso : x=[1,0] , Beneficio Total = 5

Solución óptima : x=[0,1] , Beneficio Total = 100.

3. ***Tampoco con Greedy para beneficio/peso ni con ninguna estrategia*.**

No Tiene solución con la Técnica Greedy

Técnica Greedy
Ejemplo: Planificación 1 servidor

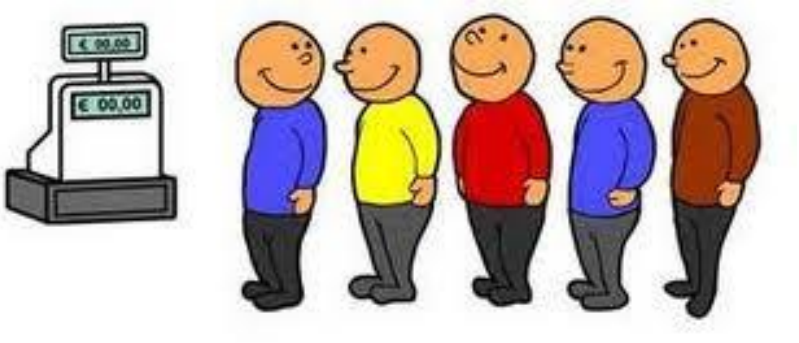
Considere ***un solo servidor*** que tiene que atender n clientes.
El tiempo de atención que necesita cada cliente se conoce de antemano.

OBJETIVO: se quiere *minimizar el tiempo medio de espera* en el sistema.

Datos:

número de cada cliente: $i=1,n$
t_i: tiempo de atención del cliente i
E_i: tiempo de espera del cliente i

Objetivo: minimizar $E = \sum_{i=1}^{n} E_i$



Cuál es la estrategia Greedy ?

Técnica Greedy
Ejemplo: Planificación 1 servidor

Problema: minimizar tiempo de espera de los clientes

Ejemplo: n=3 clientes C1, C2, C3, tiempos: t1=8, t2=4, t3=6
Tiempo total de atención para estos clientes = $\sum$ ti = 18 .
Existen n!=6 posibles esquemas de orden de atención.

	Tiempo de Espera			
ORDEN	C1	C2	C3	TIEMPO MEDIO
C1C2C3	8	8+4	8+4+6	12.7
C1C3C2	8	8+6+4	8+6	13.3
C2C1C3	4+8	4	4+8+6	11.3
C2C3C1	**4+6+8**	**4**	**4+6**	**10.7 (mínimo)**
C3C1C2	6+8	6+8+4	6	12.7
C3C2C1	6+4+8	6+4	6	11.3

El algoritmo greedy atiende a los clientes en orden creciente de *ti* y garantiza siempre la solución óptima en este problema.

Técnica Greedy
Ejemplo: Planificación vs servidores

Si se aumenta el servidores, con lo que ahora se dispone de un total de ***S*** servidores para realizar las ***n*** tareas.

En este caso también se tiene que ***minimizar el tiempo medio de espera de los clientes***, pero con la diferencia que ahora existen ***S*** servidores dando servicio simultáneamente.

Técnica Greedy
Ejemplo: Planificación vs servidores

Basándose en el método utilizado anteriormente, la forma óptima de atender los clientes es la siguiente:

- En primer lugar, se ordenan los clientes por ***orden creciente*** de tiempo de servicio.

- Una vez ordenados, se van asignando los clientes por orden, siempre al servidor menos ocupado.

- En caso de haber varios con el mismo grado de ocupación, se elige el de número menor.

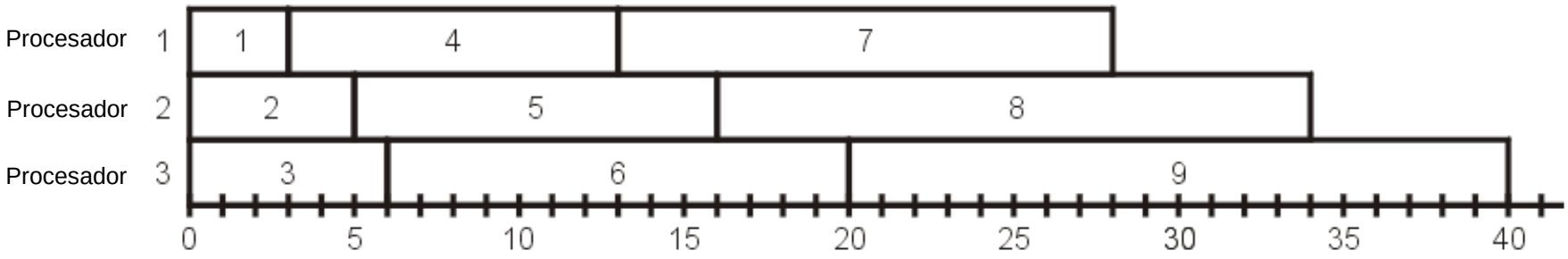
Técnica Greedy
Ejemplo: Planificación vs servidores

Si los clientes están ordenados de forma que:

$$ti \leq tj \quad \text{si} \quad i < j,$$

Un algoritmo greedy asigna al servidor k las tareas k, $k+S$, $k+2S$, ...

Por ejemplo:



Esta distribución *minimiza* el tiempo medio de espera de los clientes.