Ingeniería Informática
Algoritmos y Estructura de Datos III – Sección TQ – 1er. Semestre - 2018

Técnicas de diseño de Algoritmos (parte I)

Prof. Cristian Cappo

mayo/2018

Contenido

- Introducción
- Divide y vencerás (*divide and conquer*)
- Algoritmos voraces (*greedy*)

En la parte II (miércoles 10/mayo):

- Programación dinámica
- Vuelta atrás o Backtracking

Introducción

- Muchos problemas pueden ser resueltos aplicando algunas estrategias conocidas o "***Técnicas de diseño de algoritmos***"
- Analizaremos 4 técnicas aplicables a varias clases de problemas:
  - Divide y vencerás
  - Algoritmos voraces o *greedy algorithms*
  - Programación dinámica
  - Algoritmos de vuelta atrás o *backtracking sin/*con ramificación y poda.
- También existen otras técnicas (algunas ya vimos)
  - Fuerza Bruta: ej. búsqueda exhaustiva
  - Decrece y vencerás: alg. incrementales (ej. InsertSort)

Introducción

- Algunos problemas pueden ser resueltos con más de una técnica
- A veces ninguna técnica es posible aplicar. Por ejemplo en el caso de los problemas llamados *NP-Hard* y/o *NP-Completo* (*para este tipo de problemas a veces es solo posible aproximaciones antes de encontrar la solución exacta*).
- Es importante caracterizar el problema, por ejemplo si es de:
  - *Optimización (maximización o minimización)* : dar el cambio, problema de la mochila, etc.
  - *Búsqueda (exhaustiva o no)* : viajante, N-reinas.

Introducción

- A pesar de que algunos problemas pueden ser resueltos con varias técnicas, siempre habrá alguna que lo resuelva en forma más eficiente.
- Tenemos en cuenta: eficiencia y claridad en el código (solución *elegante*)
- En esta **parte I** veremos dos técnicas:
  - Divide y vencerás
  - Algoritmos voraces o *Greedy algorithms*.

I - Divide y vencerás

- Consiste en descomponer un problema de tamaño N en problemas más pequeños. A partir de la solución de los pequeños es posible construir la solución al problema completo
  (Aho, Alfred V., John E. Hopcroft y Ullman.Data Structures and Algorithms. 1987).
- Se basa en el uso de ***recursión***.
- Ya lo hemos utilizado para varios problemas:
  - Búsqueda binaria
  - MergeSort
  - QuickSort

Divide y vencerás

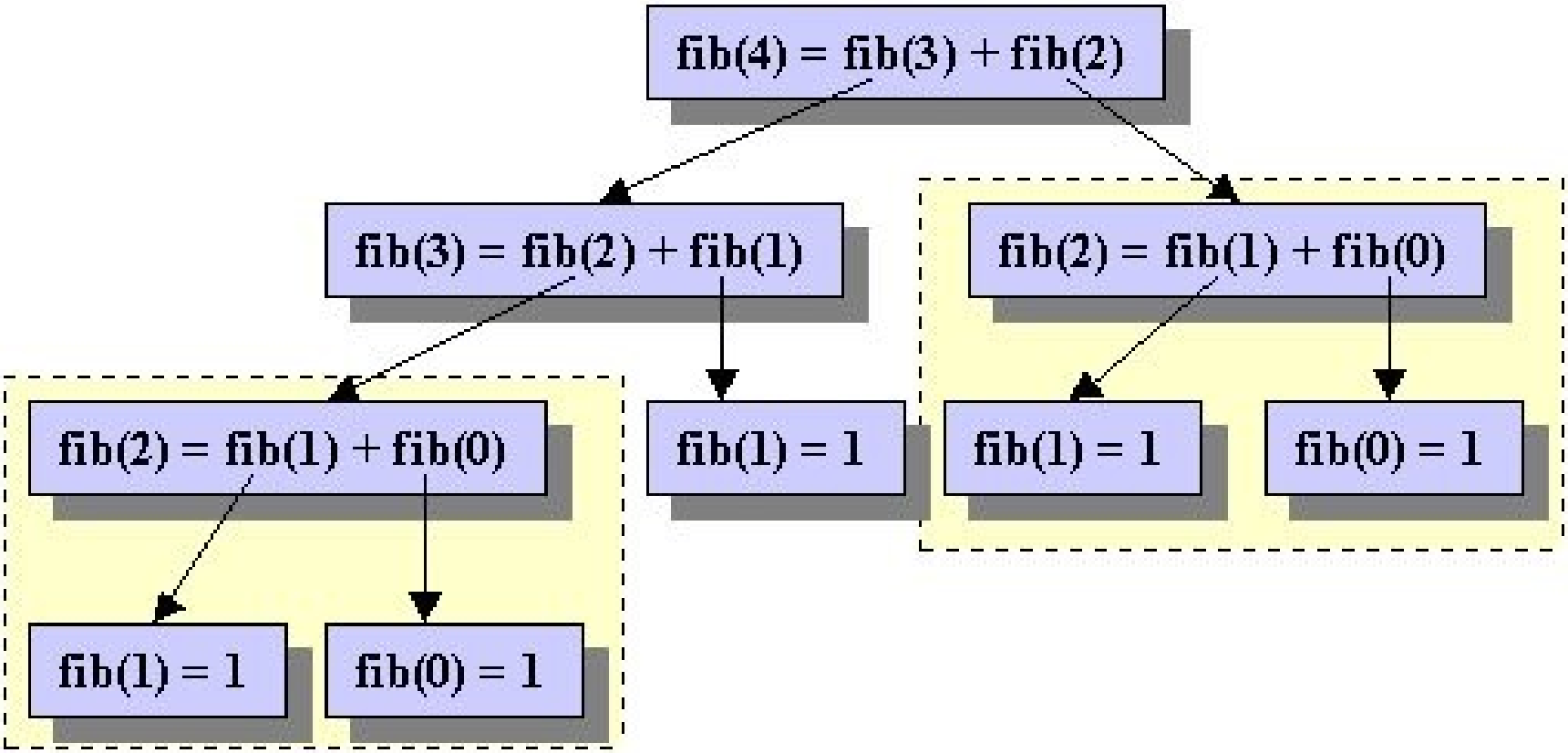
- Fundamentos de la técnica
  - **Dividir**: los problemas pequeños se resuelven recursivamente excepto el caso base. Preferible dividir en subproblemas de igual tamaño. Cada subproblema es similar al original pero de menor tamaño.
  - **Vencer**: la solución al problema original se consigue a partir de la solución de los subproblemas.
- Características
  - Al menos debe existir dos llamadas recursivas
  - Los subproblemas a resolver deben ser disjuntos. ***¿Qué significa esto?***

Divide y vencerás

- La siguiente solución, que obtiene el n-ésimo término de la serie fibonacci:

```
public static long fib ( long n ) {
    if ( n == 1 || n == 2 ) {
        return 1;
    else
        return fib(n-1) + fib (n-2);
}
```

¿Los subproblemas que resuelven son disjuntos?


fib(4) = fib(3) + fib(2)
fib(3) = fib(2) + fib(1)
fib(2) = fib(1) + fib(0)
fib(2) = fib(1) + fib(0)
fib(1) = 1
fib(1) = 1
fib(0) = 1
fib(1) = 1
fib(0) = 1

Divide y vencerás

- Según Cormen, Leiserson, Rivest y Stein (CLRS).

  Esta técnica o modelo envuelve tres pasos en cada nivel de recursión:

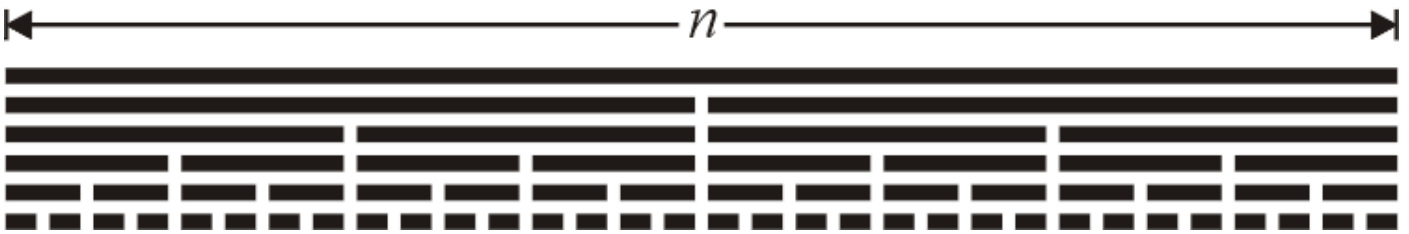
  Dividir : el problema en subproblemas

  Conquistar : los subproblemas resolviéndolos recursivamente. Si es muy pequeño (caso base) la solución es trivial.

  Combinar : las soluciones en la solución para el problema original.

Divide y vencerás

- Ejemplo clásico: mergeSort
  - Dividir la lista de tamaño n en b=2 sublistas de tamaño n/2 entradas
  - Cada sublista es ordenada recursivamente
  - Las dos sublistas son mezcladas en una sola lista ordenada



Divide y vencerás

- MergeSort:

  Más formalmente :

  - Dividir en *b* subproblemas, cada uno de tamaño *n/b* aproximadamente.
  - Resolver *a* >= *1* de estos subproblemas recursivamente.
  - Combinar las soluciones para obtener la solución completa.

Divide y vencerás

- Viendo los problemas de mergeSort y Búsqueda binaria:

MergeSort	b=2	a=2
BinarySearch	b=2	a=1

- Los tiempos son diferentes:

MergeSort	O(n log n)
BinarySearch	O(log n)

Divide y vencerás

- Por tanto, la técnica no determina el rendimiento.
- Debemos por tanto analizar:
  - El esfuerzo requerido para dividir el problema
  - El esfuerzo requerido para combinar las soluciones de los subproblemas.

Divide y vencerás

- Para mergeSort
  - La división es rápida (encontrar la mitad) : O(1)
  - Combinar tiene un costo de O(n)
- Para quickSort
  - Dividir es costoso: O(n)
  - Una vez que las sub-listas estan ordenadas, se ha finalizado: O(1) (no hay esfuerzo de combinar)

Divide y vencerás

- Así, podemos escribir las siguientes expresiones

BinarySearch

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & n > 1 \end{cases}$$

MergeSort

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

- En general, podemos asumir que el trabajo hecho para combinar es $O(n^k)$.

Divide y vencerás

- Así, la forma general de un algoritmo DyV:
  - Divide un problema en ***b*** subproblemas
  - Recursivamente resuelve ***a*** de estos subproblemas
  - Requiere ***O(n^k)*** trabajo en cada paso

  Tiene un tiempo de ejecución:

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ a\,\mathrm{T}\left(\frac{n}{b}\right) + \mathbf{O}\left(n^k\right) & n > 1 \end{cases}$$

Asumimos que un problema de tamaño 1 esta resuelto.

Divide y vencerás

- Otro ejemplo: suponga que tiene una matriz de orden N, donde cada fila y columna esta linealmente ordenada.
- Determinar si un valor ***x*** se encuentra en ella

$$\begin{pmatrix} 2 & 4 & 5 & 8 & 9 & 11 & 14 \\ 3 & 5 & 8 & 12 & 14 & 15 & 17 \\ 6 & 8 & 9 & 15 & 18 & 20 & 21 \\ 7 & 12 & 14 & 17 & 23 & 24 & 29 \\ 10 & 14 & 16 & 20 & 24 & 27 & 30 \\ 13 & 15 & 20 & 23 & 30 & 32 & 37 \\ 15 & 17 & 21 & 25 & 33 & 35 & 40 \end{pmatrix}$$

Divide y vencerás

- Ejemplo, buscar si el 19 se encuentra, empezamos en el medio

$$\begin{pmatrix} 2 & 4 & 5 & 8 & 9 & 11 & 14 \\ 3 & 5 & 8 & 12 & 14 & 15 & 17 \\ 6 & 8 & 9 & 15 & 18 & 20 & 21 \\ 7 & 12 & 14 & 17 & 23 & 24 & 29 \\ 10 & 14 & 16 & 20 & 24 & 27 & 30 \\ 13 & 15 & 20 & 23 & 30 & 32 & 37 \\ 15 & 17 & 21 & 25 & 33 & 35 & 40 \end{pmatrix}$$

Divide y vencerás

- 17 < 19, y por tanto, solo podemos excluir la submatriz superior izquierda

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Divide y vencerás

- Así, buscamos recursivamente en cada tres de las cuatro submatrices
- Cada submatriz tiene aproximadamente n/2 x n/2

Divide y vencerás

- Así, la recurrencia debe ser:

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ 3\,\mathrm{T}\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(1) & n > 1 \end{cases} \qquad Es\ n^{\log_2 3}$$

  - $\mathrm{T}(n)$ es el tiempo de búsqueda en la matriz de n x n
  - La matriz es dividida en 4 submatrices de $\frac{n}{2}$ x $\frac{n}{2}$
  - Buscamos en **3** submatrices
  - En cada paso, solo se necesita comparar el valor central: O(1).
    - Note que NO es $T(n) = 3\,T(\frac{n}{4}) + O(1)$

Divide y vencerás

- Forma general de resolución (Teorema maestro)

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ a\,\mathrm{T}\left(\frac{n}{b}\right) + cn^k & n > 1 \end{cases}$$

Donde c y k son números reales, n, a y b son números naturales y donde a>0, c>0, k>= 0 y b > 1

a = representa número de subproblemas

$\frac{n}{b}$ = representa el tamaño de cada subproblema, resuelto en tiempo $T(\frac{n}{b})$

$c\,n^k$ = costo de descomponer el problema inicial y combinar las soluciones

Divide y vencerás

- Los costos asintóticos son los siguientes (Teorema maestro o *Master Theorem*)

$$\mathrm{T}(n) \in \begin{cases} O(n^k) & \text{Si } a < b^k \\ O(n^k \log n) & \text{Si } a = b^k \\ O(n^{\log_b a}) & \text{Si } a > b^k \end{cases}$$

Divide y vencerás

Forma general de la solución DyV

```
PROCEDURE DyV(x:TipoProblema):TipoSolucion;
VAR
   i,k           :CARDINAL;
   s             :TipoSolucion;
   subproblemas  :ARRAY OF TipoProblema;
   subsoluciones:ARRAY OF TipoSolucion;
BEGIN
   IF EsCasobase(x) THEN
      s:= resolverCasoBase(x)
   ELSE
      k:= dividir(x,subproblemas);

      FOR i:=1 TO k DO
        subsoluciones[i]:=DyV(subproblemas[i])
      END;
      s:= combinar(subsoluciones)
   END;
   RETURN s;
END DyV;
```

Divide y vencerás

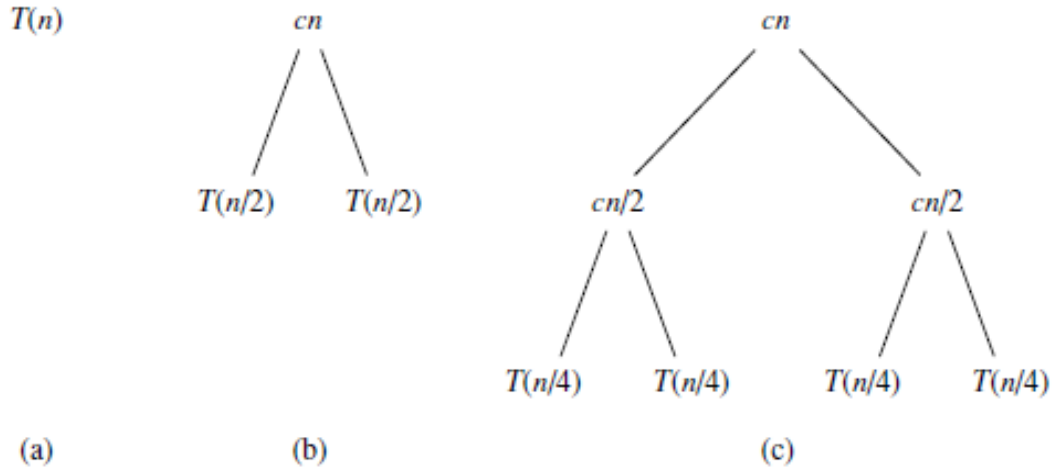
MergeSort (análisis)

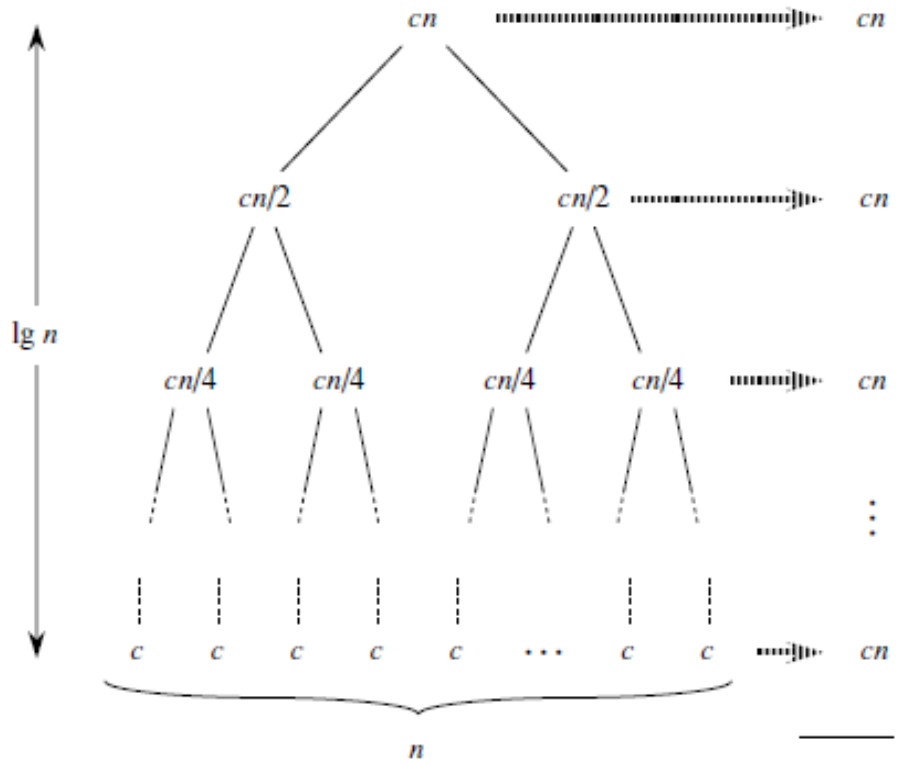
```
MERGE-SORT (A, p, r )
if ( p < r ) begin
    q = (p+r) /2
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
End
```

$$T(n) = \begin{cases} c & \text{si } n = 1 \rightarrow \text{caso base} \\ 2\,T(n/2) + cn & \text{si } n > 1 \end{cases}$$

$A = 2 \quad b = 2 \quad k = 1 \quad (c.n^1)$

Por tanto $O(n^k \log_b n) \rightarrow O(n \log n)$





Total: $cn \lg n + cn$

Otros problemas

- Considere multiplicar dos números

$$A = a_1 a_2 .. a_n \qquad a_i, b_i \in \{0,1,2,..,9\}$$
$$B = b_1 b_2 .. b_n$$

Multiplicación tradicional $A.B = O(n^2)$

$$A = A_1 | A_2$$
$$B = B_1 | B_2$$

$$A.B = A_2 B_2 + (A_1 B_2 + A_2 B_1).10^{\frac{n}{2}} + (A_1 B_1).10^n$$

$$\rightarrow A.B = O(n^2)$$

Multiplicación

$$A.B = A_2 B_2 + (A_1 B_2 + A_2 B_1).10^{\frac{n}{2}} + (A_1 B_1).10^{n}$$

```
función multiplica (A,B,n){                          Eficiencia
  if (n es pequeño) {                                   O(1)
     return A*B;                                        O(1)
  } else {
     Obtener A1, A2, B1, B2;                            O(n)
     C1  = multiplica (A2, B2, n/2);                    T(n/2)
     C2  = multiplica (A1, B2, n/2);                    T(n/2)
     C3  = multiplica (A2, B1, n/2);                    T(n/2)
     C4  = multiplica (A1, B1, n/2);                    T(n/2)
     aux = suma(C2,C3);                                 O(n)
     C4  = desplaza_izq(C4,n);                          O(n)
     aux = desplaza_izq(aux,n/2);                       O(n)
     C   = suma(C1,aux);                                O(n)
     C   = suma(C,C4);                                  O(n)
     return C;                                          O(1)
  }
}
```

$$T(n) = 4\mathrm{T}\left(\frac{n}{2}\right) + n \in O(n^2)$$

Multiplicación

$$A.B = (A_2 B_2) + [(A_1 - A_2)(B_2 - B_1) + A_1 B_1 + A_2 B_2].10^{\frac{n}{2}} + (A_1 B_1).10^n$$

```
función multiplica (A,B,n){                    Eficiencia
  if (n es pequeño) {                             O(1)
     return A*B;                                  O(1)
  } else {
     Obtener A1, A2, B1, B2;                      O(n)
     s1 = suma(A1,-A2);                           O(n)
     s2 = suma(B2,-B1);                           O(n)
     aux = multiplica(s1, s2, n/2);              T(n/2)
     C2  = multiplica(A2, B2, n/2);              T(n/2)
     C1  = multiplica(A1, B1, n/2);              T(n/2)
     aux = suma(aux,C1,C2);                       O(n)
     aux = desplaza_izq(aux,n/2);                 O(n)
     C1  = desplaza_izq(C1,n);                    O(n)
     C   = suma(aux,C1);                          O(n)
     C   = suma(C,C2);                            O(n)
     return C;                                    O(1)
  }
}
```

$$T(n) = 3\mathrm{T}(\frac{n}{2}) + n \in O(n^{\log_2 3}) \in O(n^{1.584})$$

Divide y vencerás

Otros ejemplos

- Búsqueda Binaria
- Ordenación
- Problema de selección (por ejemplo: mediana)
- Exponenciación rápida
- Multiplicación de matrices (algoritmo de Strassen)
- Subsecuencia de suma máxima
- FFT (Transformada Rápida de Fourier)
- etc, etc

Ejercicio 1 (10')

James W. Cooley (IBM)

John W. Tukey (Princeton)

Considere el siguiente código que calcula la FFT (Fast Fourier Transform)

```
public class FFT {
  public static Complex[] fft(Complex[] x) {
     int N = x.length;
     if (N == 1) return new Complex[] { x[0] };
     if (N % 2 != 0) {
        throw new RuntimeException("N no es potencia de 2");
     }

     Complex[] par = new Complex[N/2];
     for (int k = 0; k < N/2; k++)
         par[k] = x[2*k];

     Complex[] q = fft(par);
     Complex[] impar  = par;

     for (int k = 0; k < N/2; k++)
          impar[k] = x[2*k + 1];

     Complex[] r = fft(impar);
     Complex[] y = new Complex[N];

     for (int k = 0; k < N/2; k++) {
         double kth = -2 * k * Math.PI / N;
         Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
         y[k]       = q[k].plus(wk.times(r[k]));
         y[k + N/2] = q[k].minus(wk.times(r[k]));
     }
     return y;
  }
}
```

RESPONDER y resolver

a) ¿Es un algoritmo DyV? ¿Porqué?
b) ¿Cuál es la ecuación de recurrencia?
c) ¿Cuál es el costo temporal y espacial?

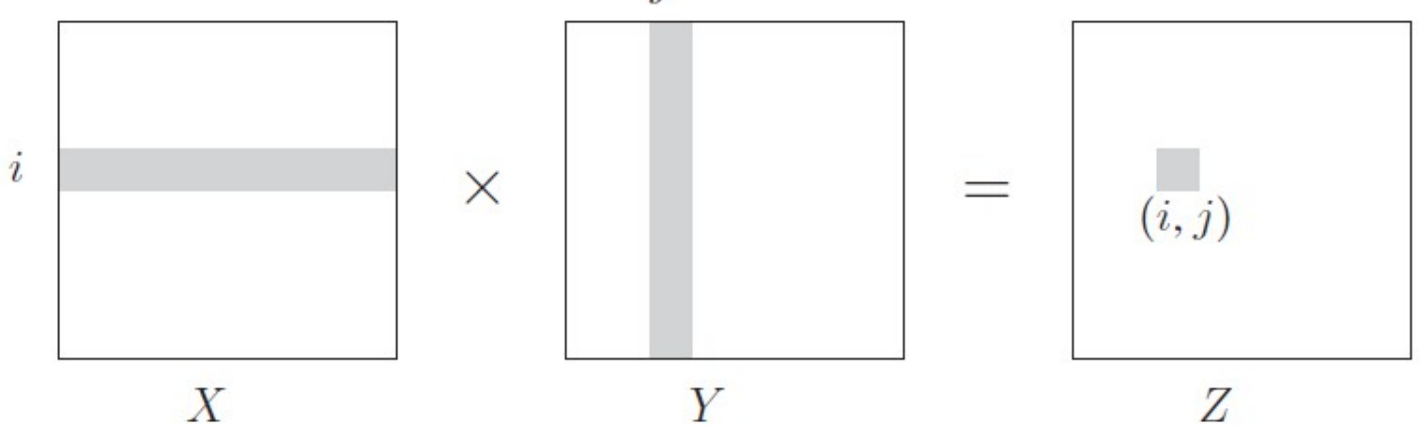
Ver un poco de historia y uso de la FFT en el libro *Algorithms* de Dasgupta, Papadimitriou y Vazirani (cap 2 – 2.6) Accesible en este link

Algoritmo presentado en "*An algorithm for the machine calculation of Complex Fourier Series*", Cooley J. & Tukey J. 1965. Mathematics of computation. 19(90) 297-301.
((uno de los algoritmos más citados, 14008 según scholar.google.com, 13438 según academic.microsoft.com))

Multiplicación de matrices

- El producto de dos matrices *n* x *n,* X e Y es una matriz *n* x *n* Z = XY, donde cada entrada *(i,j)* de Z

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$



a) Mostrar el algoritmo de multiplicación de matrices

b) Indique su tiempo de ejecución

Ejercicio 2 (10')

- Considere el mismo problema de multiplicar dos matrices de *n* x *n*:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Podemos ahora usar la estrategía divide y vencerás y encontrar recursivamente El producto XY. Esto es: solucionar 8 productos de n/2

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

a) ¿Cuál es la ecuación de recurrencia?.
b) ¿Cuál es el tiempo de ejecución?

Ejercicio 3 (10')

Volker Strassen

- Volker Strassen (1936) descubrió en 1969 el siguiente método para multiplicar dos matrices.

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Donde:

$$P_1 = A(F - H)$$
$$P_2 = (A + B)H$$
$$P_3 = (C + D)E$$
$$P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H)$$
$$P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

a) ¿Cuál es la ecuación de recurrencia?.
b) ¿Cuál es el tiempo de ejecución?

Strassen, Volker, *Gaussian Elimination is not Optimal*, Numer. Math. 13, p. 354-356, 1969

Ejercicio 4 (10')

- ¿Cuántas multiplicaciones y sumas lleva el algoritmo de multiplicación tradicional de matrices vs. el algoritmo de Strassen?

Table 5.3. Comparison between Strassen's algorithm and the traditional algorithm.

	n	Multiplications	Additions
Traditional alg.	100	$1,000,000$	$990,000$
Strassen's alg.	100	$411,822$	2,470,334
Traditional alg.	1000	$1,000,000,000$	$999,000,000$
Strassen's alg.	1000	$264,280,285$	1,579,681,709
Traditional alg.	10,000	10^{12}	9.99×10^{12}
Strassen's alg.	10,000	0.169×10^{12}	10^{12}

Extraído de , Alsuwaivel, M.H., *Algorithms. Design Techniques and Analysis*, Lectures Notes Series on Computing. World Scientific. 2016

II - Algoritmos ávidos (*Greedy Algorithms*)

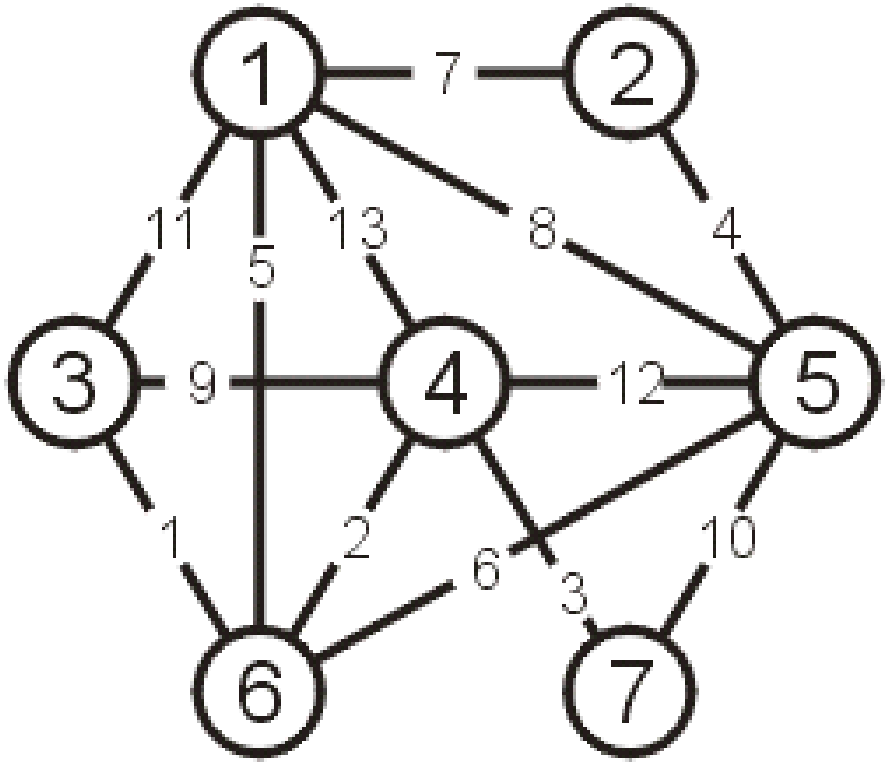
- Se basan en usar la mejor solución para un momento dado, para luego obtener la mejor solución global.

  No siempre un algoritmo ávido obtiene la mejor solución

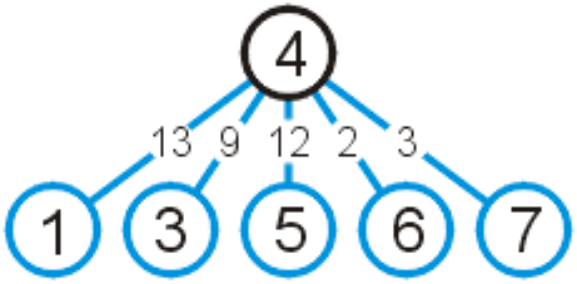
- Algunos algoritmos conocidos basados en esta técnica:
  - Algoritmo de PRIM (MST)
  - Algoritmo de KRUSKAL (MST)
  - Algoritmo de DIJKSTRA (camino mínimo)
  - Codificación de HUFFMAN (lo miraremos en detalle)
  - Problema del cambio

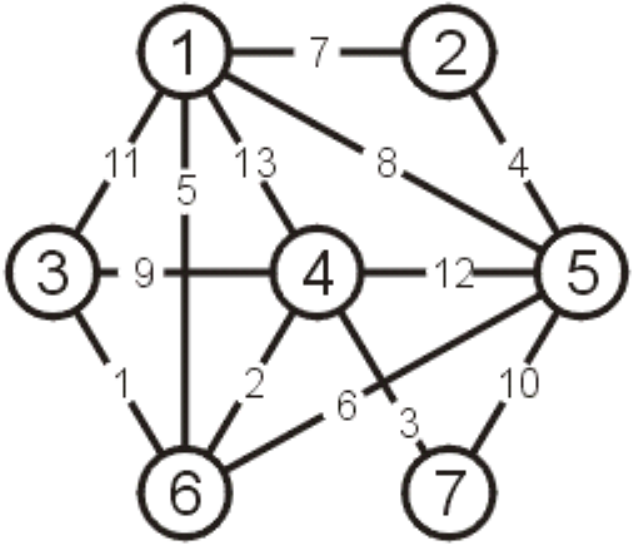
Algoritmo ávidos

- Considere el algoritmo de Prim y el siguiente grafo con pesos

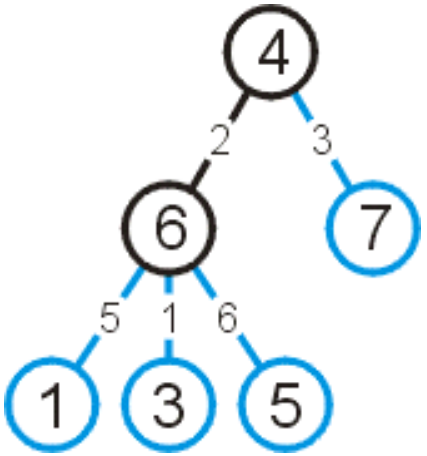


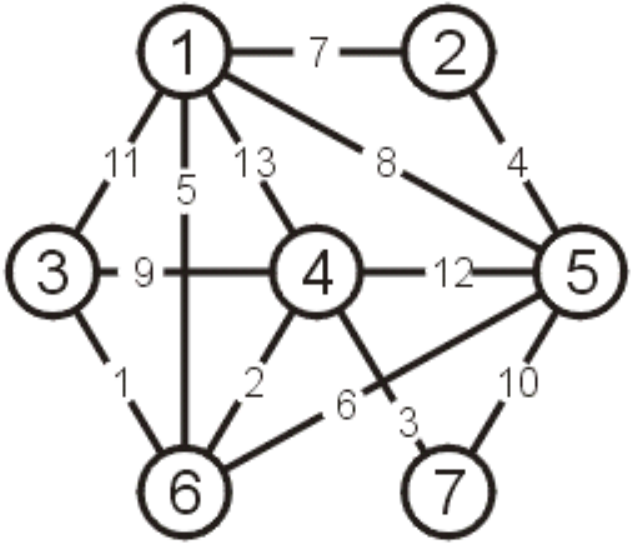
- Empezamos por un vértice cualquiera, por ejemplo el 4, que será la raíz de nuestro MST (Minimum Spanning Tree)
- Esta solución parcial no es una solución ya que no es un MST (a no ser que |V| = 1)
  - Dada esta solución parcial, la expandimos considerando todas las posibles aristas que puede tener nuestro árbol. Aquí utilizamos el concepto voraz, y tomamos el de menor costo (arista (4,6))



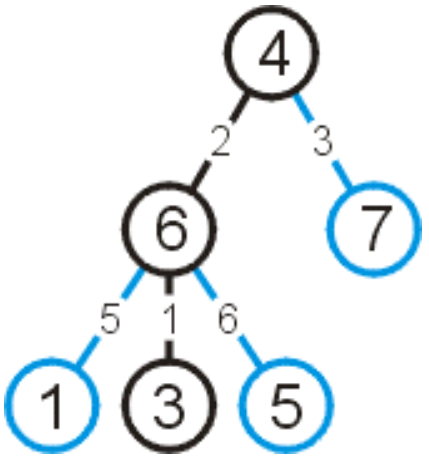


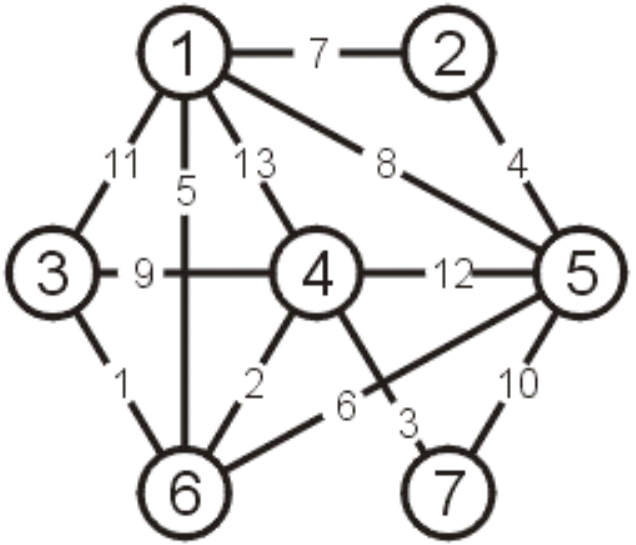
- Desde este árbol expandido, consideramos nuevamente todas las posibles aristas de expansión y de vuelta elegimos la de menor costo, en este caso la arista (6,3)



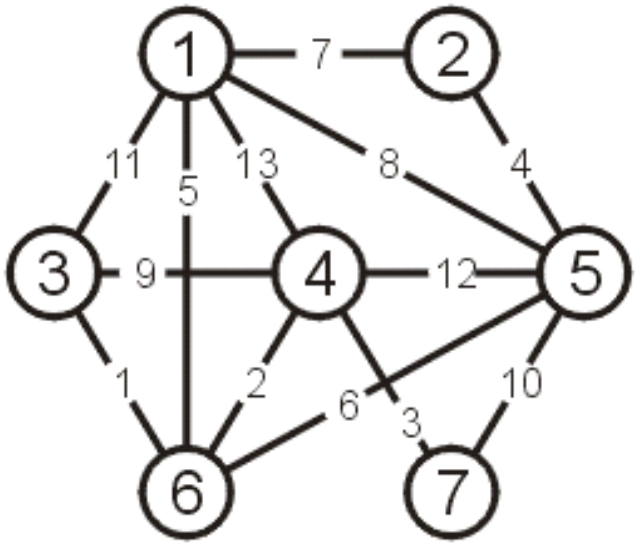


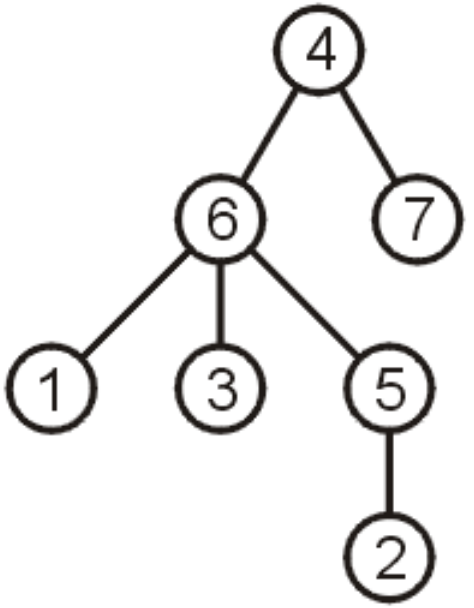
- Seguimos expandiendo el árbol, ahora consideramos la arista 4,7 que es la menor





- Luego de 6 pasos, hemos encontrado la solución, esto es, un árbol de expansión.
- En este caso encontramos un árbol de expansión que también es la solución óptima





Algoritmos ávidos

- El funcionamiento de esta técnica sigue los siguientes pasos:
  - Para cada paso, se elige la mejor opción, si se tiene
  - Se comprueba si la opción podría llevar a la solución
  - Si no puede llevar a la solución, se elige la siguiente mejor opción
  - Si el conjunto de mejores opciones escogidas es la solución, se termina el algoritmo, sino se continua.

Algoritmos ávidos

- Elementos presentes en el problema
  - Un conjunto de candidatos, que corresponden a las entradas del problema
  - Una función de selección para determinar si el candidato es idóneo
  - Una función que compruebe si un cierto conjunto es prometedor
  - Una función objetivo que determine el valor de la solución hallada
  - Una función que compruebe si el subconjunto es la solución al problema, sea óptima o no.

Algoritmos ávidos

Esquema general

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x     :ELEMENTO;
   Solucion  :CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE;
   crear(solucion); //Inicialmente vacio

   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

Algoritmos ávidos

- Codificación de Huffman*
  - Utilizado para compresión de datos, con rendimientos del 20% al 90% dependiendo de las características de los datos.
  - Utiliza una tabla de frecuencias de los caracteres para construir una representación binaria óptima de cada caracter.

* HUFFMAN, David A. *A method for the construction of minimum redundancy codes*. In Proc. IRE 40 (1951), 1098-1101

Codificación de Huffman

Suponga que tiene un archivo de 100.000 caracteres que necesitamos guardar compactado, con las siguientes frecuencias

	a	b	c	d	e	f
Frecuencia en miles	45	13	12	16	9	5
Long. Fija	000	001	010	011	100	101
Long. Variable	0	101	100	111	1101	1100

Si utilizamos codificación fija, se necesitan 3 bits (*requiere **300000 bits***)

Pero si usamos la codificación variable, podríamos aumentar la compresión. (este esquema requiere ***224000 bits***, que es aprox. 25% de ahorro). El detalle es que debemos asegurar que la decodificación se hará en forma no ambigua.

Codificación de Huffman

- La idea es encontrar una representación binaria óptima que permita la codificación y decodificación de forma no ambigua

Algoritmo

C es un conjunto de ***n*** caracteres y cada carácter ***c*** ∈ ***C*** es un objeto con un frecuencia definida en ***f[c]***

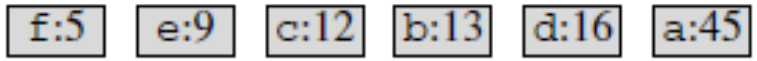
```
n = |C|  // cantidad de elementos
Q =  C // construye el heap minimal en base a los elementos de C
for i = 1 to n-1 do {
  z = new Node()
  z.left  = x  = extraer_min(Q)
  z.rigth = y  = extraer_min(Q)
  f[z] = f[x] + f[y]
  inserta(Q,z)
}
return extraer_min(Q)
```

Node

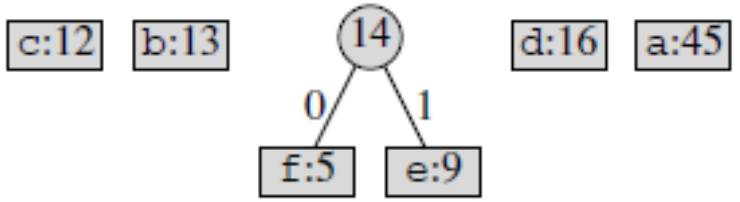
Object	data
Node	left
Node	rigth

	a	b	c	d	e	f
Frecuencia en miles	45	13	12	16	9	5

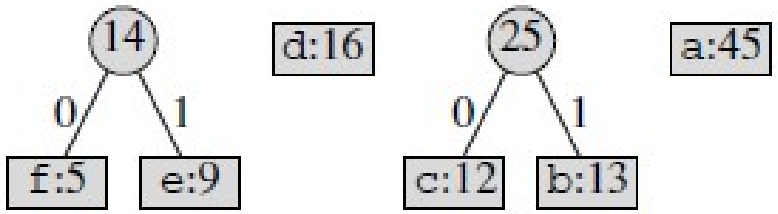
(a)



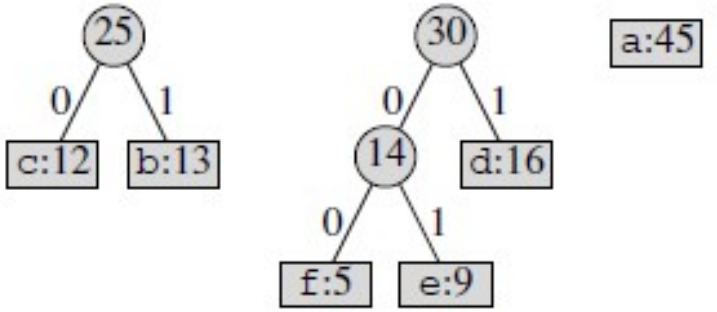
(b)



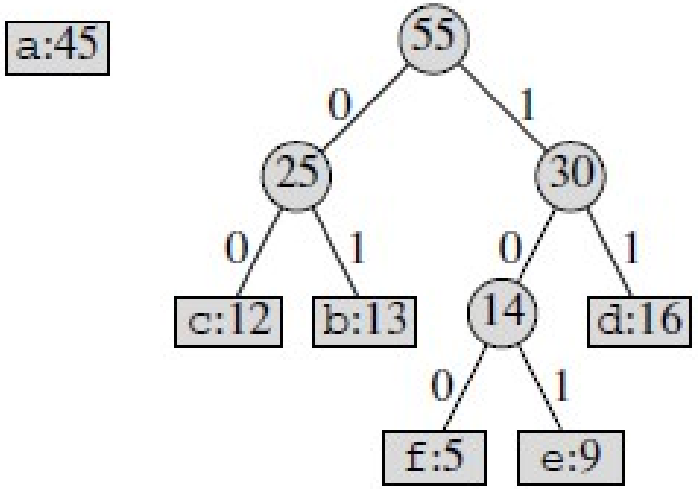
(c)



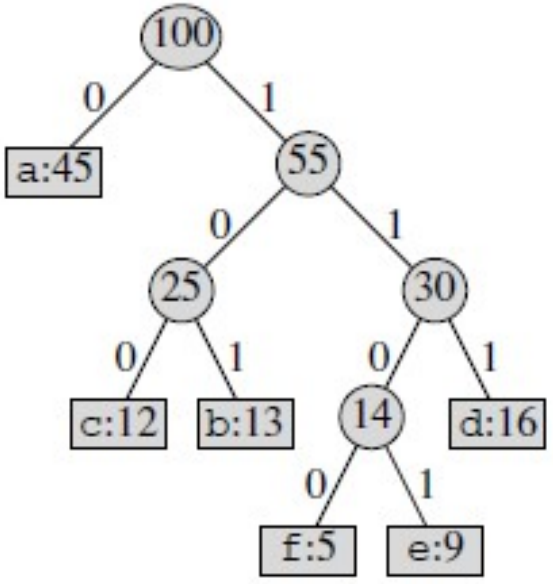
(d)



(e)



(f)



Ejercicio 5 (10')

Construya en su cuaderno la codificación de Huffman para el siguiente conjunto de letras y frecuencias:

Letra:	C	D	E	F	K	L	U	Z
Freq.:	32	42	120	24	7	42	37	2

Otro ejemplo:

Administración de procesos

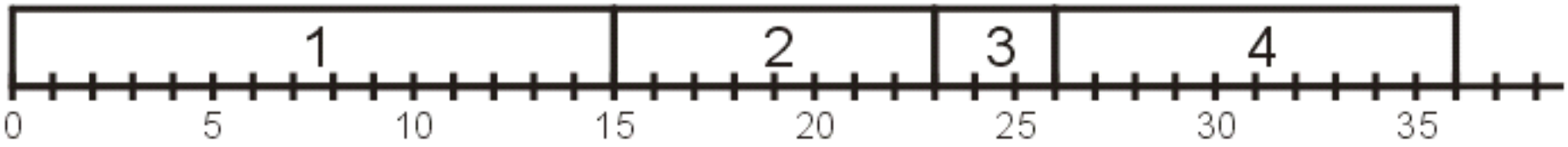
Suponga que tiene ***N*** procesos con tiempos conocidos los que deben ser ejecutados en un solo procesador

Usted debe minimizar el tiempo total de espera para que los procesos se completen

Considere el siguiente ejemplo

Proceso (i)	Tiempo (t_i)
1	15 ms
2	8 ms
3	3 ms
4	10 ms

- Si la planificación la hacemos de acuerdo al número de proceso obtendremos el siguiente esquema de procesamiento

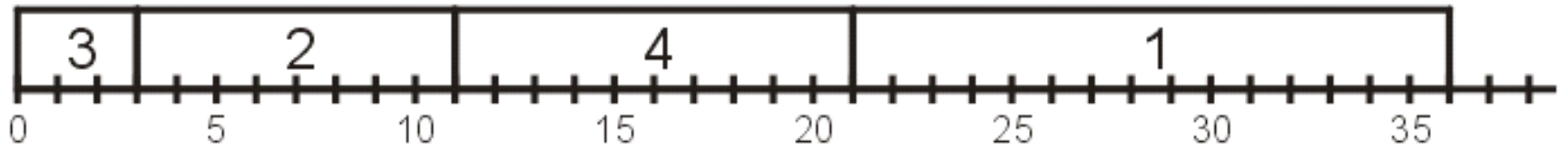


El tiempo total de espera es:

15 + 23 + 26 + 36 = 100 ms

Proceso (i)	Tiempo (t_i)
1	15 ms
2	8 ms
3	3 ms
4	10 ms

- Lo que se obtuvo no es óptimo
- Ahora ejecutemos un algoritmo voraz el cual toma el proceso con menos tiempo de procesamiento, así obtenemos el siguiente esquema de ejecución:



- El tiempo total de espera es

  3 + 11 + 21 + 36 = 71 ms

Proceso (i)	Tiempo (t_i)
1	15 ms
2	8 ms
3	3 ms
4	10 ms

- En este caso el algoritmo ávido nos da la solución óptima. ☺
- Esto podemos demostrarlo matemáticamente.
- Dado i_1, i_2, i_3, i_4 ser la permutación de los números de proceso {1, 2, 3, 4}
- El tiempo de proceso por cada uno de los procesos siguientes es:

Proceso	Tiempo
i_1	t_{i_1}
i_2	$t_{i_1} + t_{i_2}$
i_3	$t_{i_1} + t_{i_2} + t_{i_3}$
i_4	$t_{i_1} + t_{i_2} + t_{i_3} + t_{i_4}$

- El tiempo total es por tanto dada por la siguiente fórmula:

$$\sum_{k=1}^{4}(N-k+1)t_{i_k}$$

- Que puede ser reescrito así:

$$(N+1)\sum_{k=1}^{4}t_{i_k}-\sum_{k=1}^{4}kt_{i_k}$$

- Para minimizar el tiempo total, nosotros solo necesitamos maximizar la suma:

$$\sum_{k=1}^{4} kt_{i_k}$$

En nuestro orden, si $t_{i_j} < t_{i_k}$ entonces

$jt_{i_j} + kt_{i_k} > kt_{i_j} + jt_{i_k}$ ssi, $j < k$

Ejemplo:
Dado $t_1 < t_2 < t_3 < t_4$ y $j=2$ y $k=4$ entonces $2t_2 + 4t_4 > 2t_4 + 4t_2$

Consecuentemente, la ordenación óptima debe encontrarse tomando primero el proceso con menor tiempo.

- Un problema que no puede ser resuelto usando un algoritmo voraz es minimizar el tiempo total de terminación:

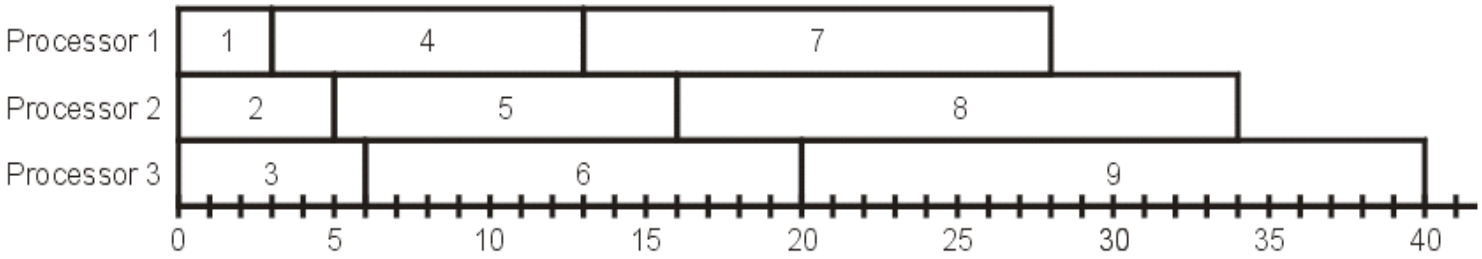
  Dado N procesadores, minimizar el tiempo total requerido para completar todos los procesos.

- Esta es una clase de problemas de la clase denominados *NP-completo*.

- Por ejemplo, queremos ejecutar estos procesos en tres procesadores diferentes

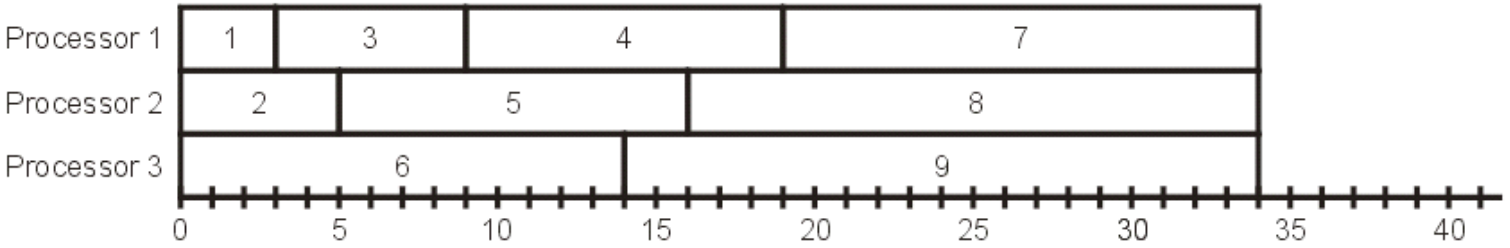
Proceso	Tiempo
1	3 ms
2	5 ms
3	6 ms
4	10 ms
5	11 ms
6	14 ms
7	15 ms
8	18 ms
9	20 ms

Si queremos minimizar el tiempo promedio de espera, asignamos cíclicamente los procesadores



Tiempo total = 165 ms

Si queremos minimizar el tiempo total de finalización, entonces tenemos el siguiente esquema:



Tiempo total = 168 ms ***¿Cual es la diferencia?***

Bibliografía

- Mark Allen Weiss. *Estructura de Datos en Java*. 4ta. Edición 2013.
- Aho, HopCroft & Ullman. *Estructura de datos y algoritmos*. 1998.
- Clifford A. Shaffer. *A practical Introduction to data structures and algorithm analysys*. Java Edition. 2013.
- Cormen, Leiserson, Rivest & Stein. *Introduction to Algorithm*. 2001 (2ra. Ed.) y 2009 (3ra. Ed)
- Rosa Guerequeta y Antonio Vallecillo. *Técnicas de diseño de algoritmos*. . Universidad de Málaga. Acceso.
- Alsuwaivel, M.H., *Algorithms. Design Techniques and Analysis*, Lectures Notes Series on Computing. World Scientific. 2016
- S. Dasgupta, C. Papadimitriou y U. Vazirani. *Algorithms*. MgHill. 2008. Acceso.