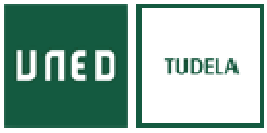

ESQUEMAS PROGRAMACIÓN III

Iñaki Alzórriz Armendáriz
Profesor-Tutor Centro Asociado de Tudela

Curso 2000 / 2010

METODO GENERAL DE RESOLUCIÓN

Los pasos que debemos seguir para la resolución de los problemas son los siguientes:

1- ***Elección del esquema***: Dar razones de por que se elige. En caso de que haya mas de un esquema aplicable escoger aquel que se aplique de forma natural al problema, y en todo caso el de coste menor.

2- ***Identificación del problema con el esquema elegido***: Partiendo del esquema general del algoritmo ver como se particularizan para nuestro caso cada una de las funciones que forman parte del esquema.

3- ***Estructura de datos***: Se menciona la estructura de los ejemplares del problema y la de cualquier otro dato relevante para su resolución. No es necesario entrar en detalles de su implementación.

4- ***Algoritmo completo***: En la medida de lo posible, no se reescribirá el esquema para ajustarlo al problema.

5- ***Estudio del coste***.

TEMA 3 - NOTACIÓN ASINTÓTICA

REGLA DEL LÍMITE

Nos permite comparar dos funciones en cuanto a la notación asintótica. Hay que calcular:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Tenemos tres resultados posibles:

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in R$

$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
$g(n) \in O(f(n))$	$g(n) \in \Omega(f(n))$	$g(n) \in \Theta(f(n))$

2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

$f(n) \notin O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \notin \Theta(g(n))$
$g(n) \in O(f(n))$	$g(n) \notin \Omega(f(n))$	$g(n) \notin \Theta(f(n))$

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) \in O(g(n))$	$f(n) \notin \Omega(g(n))$	$f(n) \notin \Theta(g(n))$
$g(n) \notin O(f(n))$	$g(n) \in \Omega(f(n))$	$g(n) \notin \Theta(f(n))$

TEMA 4 - ANÁLISIS DE ALGORITMOS

RESOLUCIÓN DE RECURRENCIAS

a)- Resolución por sustracción:

$$T(n) = \begin{cases} c * n^k & si\ 0 \leq n < b \\ a * T(n-b) + c * n^k & si\ n \geq b \end{cases}$$

La resolución de esta ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & si\ a < 1 \\ \theta(n^{k+1}) & si\ a = 1 \\ \theta(a^{n\ div\ b}) & si\ a > 1 \end{cases}$$

b)- Resolución por división:

$$T(n) = \begin{cases} c * n^k & si\ 0 \leq n < b \\ a * T\left(\frac{n}{b}\right) + c * n^k & si\ n \geq b \end{cases}$$

La resolución de esta ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & si\ a < b^k \\ \theta(n^k * \log(n)) & si\ a = b^k \\ \theta(n^{\log_b a}) & si\ a > b^k \end{cases}$$

ORDEN EXACTO: t(n) está en orden exacto de f(n) y escribiremos t(n) ϵ Θ(f(n)) si t(n) pertenece tanto a O(f(n)) cota superior, como a Ω(f(n)) cota inferior.

$$\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

TEMA 5 - ESTRUCTURAS DE DATOS

MONTICULOS

Propiedad de montículo:

- El nodo i es el padre de (2*i) y de (2*i + 1)
- El nodo i es el hijo de (i div 2)

Esto para los vectores

-Hijos con respecto a los padres: T[i] ≥T[2*i] y T[i] ≥ T[2*i + 1]

- Padres con respecto a hijos: T[i] ≤ T[i div 2]

Tenemos dos posibles algoritmos para crear un montículo:

1ª Opción para crear un montículo (opción "lenta"):

```
proc crear-monticulo-lento (T[1..n])
    para i ←2 hasta n hacer flotar(T[1..i],i)
fproc
```

Algoritmo iterativo para flotar una posición:

```
proc flotar (T[1..n],i)
    k ←i
    repetir
        j←k
        si j>1 y T[j div 2]<T[k] entonces k←j div 2
        intercambiar T[j]y T[k]
        {si j=k entonces el nodo ha llegado a su posición final}
    hasta que j=K
fproc
```

Estos dos algoritmos los podemos hacer de forma recursiva:

```
proc crear-monticulo-lento-recursivo  (T[1..n],i)
    si i>n entonces
        devuelve(T)
    sino
        flotar-recursivo(T,i)
        crear-monticulo-lento-recursivo(T,i+1)
    fis
fproc

proc flotar-recursivo (T[1..n],i)
    i_padre ←i div 2;
    si (i>1) and T[i]>T[i_padre] entonces
        intercambiar(T[i],T[i_padre])
        flotar-recursivo(T,i_padre)
    fsi
fproc
```

2ª Opción para crear un montículo (opción "rápida"):
(este es más eficiente, menor constante multiplicativa)

```
proc crear-monticulo (V[1..n])
    para i ← ⌊n/2⌋ bajando hasta 1 hacer hundir(T,i)

fproc
```

(NOTA: Nos refererimos al "suelo" de n/2).

Vamos a ver el algoritmo para hundir un elemento del montículo:

```
proc hundir (T[1..n],i)
        k ←i
        repetir
                j ← k
                {buscar el hijo mayor del nodo j}
                si 2*j ≤ n y T[2*j] > T[k]entonces k ← 2*j
                si 2*j < n y T[2*j + 1] > T[k] entonces k ← 2*j + 1
                intercambiar T[j] y T[k]
                {si j=k entonces el nodo ha llegado a su posición final}
        hasta que j=k
fproc
```

Podemos pensar también en una versión recursiva de este procedimiento:

```
proc hundir-recursivo (T[1..n],i)
        hmayor ←i
        {buscar al hijo mayor del nodo i}
        si (2*i≤n) y (T[2*i]>T[hmayor]) entonces hmayor←2*i;
        si (2*i<n) y (T[2*i+1]>T[hmayor]) entonces hmayor←2*i+1;
        si (hmayor>i) entonces
                intercambiar(T[i],T[hmayor]);
                hundir-recursivo(T,hmayor);
        fsi
fproc
```

Esto es considerando un ***montículo de máximos*** (la raíz es el máximo elemento).
Para eliminar la raíz haremos una vez obtenida T[1]←T[n] y haremos hundir(T[1..n],1)

Costes de un montículo:

- Construcción de un montículo: lineal.
- Ordenar n elementos: O(n log n)

Para ***añadir un nodo*** al montículo se añade en la posición n+1 del vector y posteriormente "flotamos" esa posición:

```
proc añadir-nodo(T[1..n],v)
        T[n+1]← v
        flotar(T[1..n+1] , n+1)
fproc
```

TEMA 6 - ALGORITMOS VORACES

6.1 – ESQUEMA BASICO

Se suelen usar para solucionar problemas de optimización. En estos algoritmos nunca se reconsidera una decisión.

```
fun voraz (C:conjunto) dev (S:conjunto)
    S ← Ø
    mientras ¬ solución(S) ^ C ≠ Ø hacer
        x ← seleccionar(C ) //elemento que maximiza objetivo(x)
        C ← C \ {x}
        si completable (S U {x})
            entonces S ← S U {x}
        fsi
    fmientras
    dev S
ffun
```

Funciones a particularizar: *solución*(C), *seleccionar* (C) y *completable*( )

6.2 – ARBOLES DE RECUBRIMIENTO MINIMO

El problema consiste en encontrar un sub-grafo de n-1 aristas que sea conexo y de coste (longitud) mínimo.

6.2.1 – Kruskal : Va eligiendo aristas de menor a mayor coste y si esa arista une dos componentes hasta entonces no conexas las añade al conjunto de soluciones.

```
fun Kruskal (G=<N,A>:grafo ; longitud ) dev (T:conjunto)
    Ordenar A por longitudes crecientes
    n ← número de nodos en N
    T ← Ø
    Iniciar n conjuntos, cada uno de los cuales contiene un elemento de N
    repetir
        e ← {u,v} arista más corta aún no considerada
        compu ← buscar(u)
        compv ← buscar(v)
        si compu ≠ compv entonces
            fusionar(compu,compv)
            T ← T U {e}
        fsi
    hasta que T contenga n-1 aristas
    devolver T
ffun
```

Coste de este algoritmo: Θ(a log n). Es decir depende del número de nodos y del número de aristas. Así, en un grafo denso su coste es Θ(n^2 log n) y en un grafo disperso Θ(n log n).
En este caso un montículo nos permite mismo coste pero no se ordenan aristas inútiles.

6.2.2 – Prim: Este parte de una raíz. En cada paso se elige la rama más corta de todas las que salen de los nodos ya seleccionados y que lleva a un nodo no seleccionado todavía.

```
fun Prim (G=<N,A>:grafo, longitud)  dev (T:conjunto)
        {inicialización}
        T ← Ø
        para i ←2 hasta n hacer
                mas próximo[i] ←1
                distmin[i] ←L[i,1]
        fpara
        {bucle voraz}
        repetir n-1 veces
                min ← ∞
                para j←2 hasta n hacer
                        si 0 ≤ distmin[j] < min entonces
                                min ← distmin[j]
                                K ← j

                        fsi
                fpara
                T ← T U {{mas próximo[k],k}}
                distmin[k] ← -1 {se añade k a B}
                para j ←2 hasta n hacer
                        si L[j,k]<distmin[j] entonces
                                distmin[k] ← L[j,k]
                                mas próximo[j] ← k
                        fsi

                fpara
        frepetir
        devolver T
ffun
```

Coste: Este algoritmo tiene siempre un coste de $\Theta(n^2)$. Por lo que dependiendo de la dispersión del grafo podrá ser mejor que el de Kruskal o no.
Si este se implementa con un montículo de mínimos, tendría un coste de Θ(a * log n) por lo que igualaría su coste al de Kruskal.

Esquema que aparece en el libro de ejerccios:

```
fun prim (g=<N,A>:grafo) dev conjunto
        S ← Ø
        B← Un elemento arbitrario de N
        mientras B≠N hacer
                Buscar e={u,v} de longitud mínima tal que uϵB y vϵN\B
                S←S U{e}
                B ←B U{v}
        fmientras
        devolver S
ffun
```

6.3 – CAMINOS MINIMOS

El algoritmo de Dijkstra halla los caminos más cortos desde un único origen hasta los demás nodos del grafo.

```
fun Dijkstra (L[1..n, 1..n]) dev (D: matriz[2..n])
        matriz D[2..n]
        C ←{2,3,…,n}
        para i ←2 hasta n hacer D[i] ←L[1,i]
        {bucle voraz}
        repetir n-2 veces
                v ←algún elemento de C que minimiza D[v]
                C ←C \ {v}
                para cada w en C hacer
                        D[w] ←min (D[w] , D[v]+L[v,w])
                fpara
        frepetir
        devolver D
ffun
```

Coste: Este algoritmo tiene coste $\Theta(n^2)$. Inicialización requiere un tiempo O(n) y el bucle del algoritmo voraz $\Theta(n^2)$, por tanto el coste del algoritmo es de $\Theta(n^2)$.
Si usamos un montículo de mínimos con un elemento para cada elemento *v* de C. En este caso la inicialización del montículo requiere un tiempo que está en Θ(n) y eliminar la raíz del montículo requiere un tiempo O(log n), estas son las operaciones más internas del algoritmo.
Si se produce la eliminación de la raíz (el bucle se ejecuta n-2 veces) y hay que flotar un máximo de a nodos entonces tenemos un tiempo total de Θ((a+n) * log n).

- Si el grafo es conexo (a ≥ n-1) el tiempo total será Θ(n * log n)
- Si el grafo es denso entonces será mejora la implementación con matrices.

Hay que saber su ***demostración*** (página 225).

6.5 – PROBLEMA DE LA MOCHILA *(Solución voraz solo si se pueden partir los objetos)*

```
fun mochila (W[1..n] , v[1..n], W) dev (x: matriz [1..n])
        para i=1 hasta n hacer x[i] ←0
        peso ←0
        {bucle voraz}
        mientras peso < N hacer
                i ← el mejor objeto restante
                si peso+W[i] ≤ W entonces
                        x[i] ←1
                        peso ← peso + W[i]
                sino
                        x[i] ←(W-peso)/w[i]
                        peso ←W
                fsi
        fmientras
        dev x
ffun
```

Para encontrar la solución optima hay que seleccionar los objetos por orden decreciente de v_i/w_i.
Coste: Θ(n) e incluyendo la ordenación Θ(n log n).

6.6 – PLANIFICACION

6.6.1 – Minimizar tiempo en el sistema: Elegiremos los candidatos en orden ascendente de tiempo requerido. Sencillo algoritmo voraz.
Coste: Θ(n log n) para la ordenación y Θ(n) para el algoritmo voraz. Por eso coste asintótico será de Θ(n log n).
Saberse la ***demostración*** de que el algoritmo voraz es óptimo (página 232).

6.6.2 – Planificación con plazo fijo : Hay que maximizar el beneficio total que nos proporciona un conjunto de tareas. Cada tarea cuesta una unidad de tiempo y tiene una ganancia g_i y un tiempo máximo d_i.

6.6.2.1 – Primera implementación: Cuando se considera la tarea i, se comprueba si puede insertar en j en el lugar oportuno sin llevar alguna tarea que ya esté en j más allá de su plazo. De ser así i se acepta, en caso contrario se rechaza.

```
fun secuencia (d[0..n]) dev (K,matriz[1..k])
        matriz j[0..n]
        d[0] ← j[0] ←0 {centinelas}
        k ←j[1]←1 {K nos dice el nº de tareas. La tarea 1 siempre se selecc.}
        {bucle voraz}
        para i ← 2 hasta n hacer {orden decereciente de g}
                r ← k
                mientras d[j[r]] > max(d[i],r) hacer   r ← r-1
                si d[i] > r entonces
                        para m ← k paso -1 hasta r+1 hacer
                                j[m+1] ←j[m]
                        fpara
                        j[r+1] ←i
                        k ←k+1
                fsi
        fpara
        devolver k, j[1..k]
ffun
```

Coste: $\Omega(n^2)$

6.6.2.2 – Segunda implementación: Se empieza con una planificación vacía de longitud n. Para cada tarea sucesivamente se planifica cada tarea i en el instante t donde t es el mayor entero tal que 1≤t≤min(n,di) y la tarea que se ejecuta en el instante t no está decidida todavía.

```
fun secuencia2 (d[1..n] ) dev (K , j: matriz [1..k])
        matriz j,F[0..n]
        p= min.(n,max{d[i] | 1 ≤ i ≤n})
        para i ←0 hasta p hacer
                j[i] ←0
                F[i] ←i
                iniciar el conjunto {i}
        fpara
        {bucle voraz}
        para i ←1 hasta n hacer {orden decreciente de g}
                k ←buscar (min(p,d[i]))
                m ←F[k]
                si m ≠ 0 entonces
                        j[m] ←i
                        l ←buscar(m-1)
                        F[k] ←F[l]
                        fusionar(k,l) {el cjto resultante tiene etiqueta k o l}
                fsi
        fpara
        {solo queda comprimir la solución}
        para i ←1 hasta p hacer
                si j[i]>0 entonces
                        k ←k+1
                        j[k] ←j[i]
                fsi
        fpara
        devolver k , j[1..k]
ffun
```

El ***funcionamiento*** es el siguiente:

- <u>*Inicialización*</u>: Toda posición 0,1,2,…p está en un conjunto diferente y F([i])=i. La posición 0 sirve para ver cuando la planificación está llena.
- <u>*Adición de una tarea con plazo d*</u>: Se busca el conjunto que contenga a d; sea K ese conjunto. Si F(K)=0 se rechaza la tarea, en caso contrario:
  - Se asigna la nueva tarea a la posición F(K)
  - Se busca al conjunto que contenga a F(k)-1. Lo llamamos L.
  - Se fusionan K y L. El valor F para este nuevo conjunto es el valor viejo de F(L).

TEMA 7 - ALGORITMOS DIVIDE Y VENCERAS

7.2 – ESQUEMA BASICO

Primeramente ponemos el esquema que viene en el libro:

```
funcion DV(x)
    si x suficientemente pequeño entonces proporcionar ad hoc(X)
    descomponer x en casos mas pequeños xy,x2,.....,xt
    para i←1 hasta t hacer yi ←DV(Xi)
    recombinar los yi para obtener la solución
    devolver y
ffun
```

Pero se ve más claro en el algoritmo general presentado en el libro de problemas:

```
fun divide-y-venceras (problema)
    si suficientemente-simple(problema) entonces
        dev solución-simple(problema)
    si no
        {p1…pk} ← descomposición (problema)
        para cada pi hacer
            si ←divide-y-venceras (pi)
        fpara
        dev combinación (Si..Sk)
    fsi
ffun
```

Para que el enfoque divide y vencerás merezca la pena tres condiciones:

1- Decisión de usar el algoritmo básico debe tomerse cuidadosamente.

2- Tiene que poderse descomponer en subejemplares y recomponer las soluciones de forma bastante eficiente.

3- Los subejemplares tiene que ser de tamaño aproximadamente igual.

Coste: Vamos a hablar siempre de una resolución de recurrencias por división

$$t(n) = lt(n\ div\ b) + g(n)$$

$$t(n) \in \begin{cases} \Theta(n^k) & si\ l < b^k \\ \theta(n^k \log n) & si\ l = b^k \\ \theta\left(n^{log_b\ l}\right) & si\ l > b^k \end{cases}$$

7.3 – BUSQUEDA BINARIA

```
funcion binrec (T[i..j],x)
        {búsqueda binaria en la submatriz T[i..j] con la seguridad de que T[i-1]<x≤T[j]}
        si i = j entonces devolver i
        k ←(i+j) div 2
        si x ≤ T[k]entonces devolver binrec (T[i..k],x)
        sino devolver binrec (T[k+1..j],x)
ffun
```

Coste: Θ(log m)

Es fácil encontrar una versión iterativa:

```
funcion biniter (T[1..n],x)
        {búsqueda binaria iterativa de x en la matriz T}
        si x > T[n]entonces devolver n+1
        i←1 ; j←n
        mientras  i < j hacer
                {T[i-1]<x≤T[j]}
                k←(i+j) div  2
                si x ≤ T[k]entonces j←k
                sino i ← k+1
        fmientras
        devolver i
ffun
```

7.4 – ORDENACIONES

7.4.1 – Ordenación por fusión:

```
Fun ordenarporfusion (T[1..n])
        si n suficientemente pequeño entonces insertar(T)
        sino
                matriz U[1.. 1 + ⌊n/2⌋] , V[1.. 1 + ⌊n/2⌋]]
                U[1.. 1 + ⌊n/2⌋] ← T[1.. 1 + ⌊n/2⌋]
                V[1.. 1 + ⌊n/2⌋] ← T[1 + ⌊n/2⌋ ..n]
                ordenarporfusion (U[1.. 1 + ⌊n/2⌋])
                ordenarporfusion(V[1.. 1 + ⌊n/2⌋])
                fusionar (U,V,T)
        ffun

        funcion fusionar (U[1..m+1] , V[1..n+1], T[1..m+n])
                i,j ←1
                U[m+1],V[n+1] ← ∞   {Centinelas}
                para k ←1 hasta m+n hacer
                        si U[i]<V[j]
                        entonces   T[k] ←U[i]; i←i+1
                        sino T[k]←V[j] ; j←j+1
                fpara
        ffun
```

Coste: Θ(n log n) . Similar al coste de ordenación por montículo.

7.4.2 – Quicksort: En el procedimiento pivote se elige un pivote, se echan los menores a la izquierda, los mayores a la derecha y el pivote se queda en medio.

```
procedimiento pivote (T[i..j] , var l)
        p ← T[i]
        k ←i ; l ←j+1
        repetir k←k+1 hasta que T[k]>p o K≥j
        repetir l ←l-1 hasta que T[l]≤ p
        mientras k<l hacer
                intercambiar T[k]y T[l]
                repetir k←k+1 hasta que T[k]>P
                repetir l←l-1 hasta que T[l]≤p
        fmientras
        intercambiar T[i] y T[l]
fprocedimiento

procedimiento quicksort  (T[i..j])
        si j-i es suficientemente pequeño entonces insertar (T[i..j])
        sino
                pivote(T[i..j],l)
                quicksort(T[i..l-1])
                quicksort(T[l+1..j])
        fsi
fprocedimiento
```

Coste: Θ(n log n) cuando el vector ya estaba ordenado. Eso si, en el peor caso tiene un coste $\Omega(n^2)$. La constante oculta de este algoritmo es menor que las asociadas a la ordenación por fusión o a la ordenación por montículo.
Si usamos *pivotebis* conseguimos coste Θ(n log n) incluso en el peor caso.

A continuación voy a incluir otros dos algoritmos de ordenación que no son divide y vencerás pero que puede ser útil conocer para responder a algunas preguntas de examen.

7.4.3 – Ordenación por inserción: Consiste en insertar un elemento dado en el lugar que le corresponde de la parte ordenada del vector.

```
procedimiento insertar (T[1..n])
        para i←2 hasta n hacer
                x←T[i] ; j←i-1
                mientras j>0 y x<T[j] hacer
                        T[j+1]←T[j]
                        j←j-1
                fmientras
                T[j+1]←x
        fpara
fprocedimiento
```

Coste: $\Theta(n^2)$

7.4.4 – Ordenación por selección: La idea es seleccionar el menor elemento de una parte desordenada y colocarlo en la primera posición del primer elemento no ordenado.

```
procedimiento selección (T[1..n])
    para i←1 hasta n hacer
        minj←i ; minx←T[i]
        para j←i+1 hasta n hacer
            si T[j]<minx entonces
                minj←j
                minx←T[j]
            fsi
        fpara
        T[minj]←T[i]
        T[i]←minx
    fpara
fprocedimiento
```

Coste: $\Theta(n^2)$

7.4.5 – Ordenación por montón: Básicamente lo que hacemos es construir un montículo en el que el elemento menor será la cima, a continuación se construye con los siguientes n-1 elementos y así sucesivamente. (Propiedades montón T[i]≥T[2*i] y T[i]≥T[2*i + 1])

```
Procedimiento ordenación-monticulo (T[1..n])
    Crear-monticulo(T)
    Para i←n bajando hasta 2 hacer
        Intercambiar T[1] y T[i]
        Hundir (T[1..i-1,1)
```

Coste: $\Theta(n \log n)$ → Hundir tiene coste $\Theta(\log n)$, en el caso pero que hay que undir todo es cuando el coste es $\Theta(n \log n)$

7.5 – BUSQUEDA DE LA MEDIANA

Si se usa el quicksort con la nueva función pivote llamada pivotebis (divide en tres partes, menores que el pivote, iguales al pivote, mayores que el pivote). Y nos vamos quedando con el trozo en que esta la posición que buscamos. Se termina cuando la posición que buscamos este en el trozo de cinta igual al pivote.

```
función selección (T[1..n],s)
    {buscar el s-ésimo elemento mas pequeño de T[1..n]}
    i←1 ; j←n
    repetir
        p←mediana(T[i..j]) {necesitamos un algoritmo eficiente para calcular la mediana}
        pivotebis(T[i..j],p,k,l)
        si s≤k entonces j←k
        sino si s≥l entonces i←l
            sino devolver p
        fsi
        fsi
    frepetir
ffun
```

Ya que la mediana difícil de calcular, la media aproximada la podemos calcular rápidamente con la media de una serie de valores al azar del array y usar esa pseudomedia como pivote.

Coste: $\Theta(n \log n)$. En el caso peor coste cuadrático, pero salvado con el cálculo de la pseudomediana. En ese caso encuentra el elemento s-esimo más pequeño en coste lineal.

7.7 – EXPONENCIACION

Forma fácil de calcular $x=a^n$ para números pequeños (de coste lineal):

```
funcion exposec (a,n)
        r←a
        para i←1 hasta n hacer
                r←r xa
        fpara
        devolver r
ffun
```

Esto para números grandes nos dispara los tiempos, ya que las multiplicaciones desbordan el tamaño de palabra y la cosa se complica, con lo que llegamos a estimar costes de $\Theta(m^2 n^2)$. La calve para aplicar el divide y veneras es que $a^n = (a^{\frac{n}{2}})^2$. Podemos entonces encontrar la siguiente ecuación de recurrencia:

```
funcion expoDV (a,n) dev entero
        si n=1 entonces devolver a
        si n es par entonces
                devolver [expoDV(a,n/2)]^2
        sino
                devolver a * expoDV(a,n-1)
        fsi
ffun
```

Ecuación de recurrencia: $N\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + 1 \leq N(n) \leq N(\left\lfloor\frac{n}{2}\right\rfloor + 2$

Por tanto a=1, b=2 y cn^k=0 por tanto el coste es $\Theta(\log n)$.

En este caso nos encontramos también con un coste $\Theta(m^2 n^2)$. Si bien es algo menor que el visto inicialmente cuando se usa el algoritmo de multiplicación divide y vencerás en vez del algoritmo clásico de multiplicación.

Podemos encontrar la versión iterativa de este algoritmo:

```
funcion expoiter(a,n)
        i←n ; r←1 ; x←a
        mientras i>0 hacer
                si i es impar entonces r←rx
                x ←x2
                i←i div 2
        fmientras
        devolver r
ffun
```

7.8 – CRIPTOGRAFIA

Para calcular: a^n mod z usaremos el siguiente algoritmo que tiene $\Theta(\log n)$

```
funcion expomod(a,n,z)
        i ← n ; r ← 1 ; x←a
        mientras i>0 hacer
                si i es impar entonces r ← (r*x) mod z
                x ← x^2 mod z
                i ← i div 2
        fmientras
        devolver r
ffun
```

TEMA 9 - EXPLORACION DE GRAFOS

9.3 – RECORRIDO EN PROFUNDIDAD

Veamos primero su algoritmo recursivo:

```
procedimiento recorridop(G)
        para cada v ϵ N hacer
                marca[v]←no visitado
        para cada v ϵ N hacer
                Si marca[v] ≠ visitado entonces rp(v)

procedimiento rp(v)
        marca[v]← visitado
        para cada nodo w adyacente a v hacer
                si marca[v] ≠ visitado entonces rp(w)
```

NOTA: Podría no encontrar solución si existen ramas infinitas. En este caso se quedaría atrapado en una rama sin encontrar solución.

Coste: Θ(max(a,n)) donde a=nº aristas y n=nº de nodos.

También hay una versión iterativa del mismo. Este se ayudará de una pila P:

```
procedimiento rp2(v)
        P ← pila vacia
        marca[v] ← visitado
        apilar v en P
        mientras P no esté vacia hacer
                mientras exista un nodo w adyacente a cima(P)
                tal que marca[w]≠visitado hacer
                        marca[w]←visitado
                        apilar w en P{w es la nueva cima}
                fmientras
                desapilar P
        fmientras
fprocedimiento
```

NOTA: Este procedimiento se llama desde el procedimiento *recorridop*(G).

El esquema de búsqueda en profundidad iterativa del libro de ejercicios:

```
fun búsqueda-en-profundadad(ensayo)
        P ←pila-vacia
        apilar(ensayo,p)
        mientras ¬vacia(p) hacer
                nodo←desapilar(p)
                si valido(nodo) entonces dev nodo
                sino
                        para cada hijo en compleciones(nodo) hacer
                                si condiciones-de-poda(hijo) entonces apilar(hijo,p) fsi
                        fpara
                fsi
        fmientras
ffun
```

NOTA: La búsqueda ciega en profundidad se diferencia de la vuelta-atrás en que esta última tiene condiciones de poda. Por tanto, podemos considerar el algoritmo anterior como una versión iterativa de vuelta-atrás ya que tiene condiciones de poda.

9.5 – RECORRIDO EN ANCHURA

El recorrido en anchura se apoya en una cola

```
Procedimiento ra(v)
    Q ← cola vacia
    marca[v]←visitado
    poner v en Q
    para cada nodo w adyacente a u hacer
        si marca[w]≠visitado entonces
            marca[w]←visitado
            poner w en Q
```

El esquema de recorrido en anchura que viene en el libro de ejercicios:

```
fun búsqueda-en-anchura(ensayo)
    p ← cola-vacia
    encolar(ensayo,p)
    mientras ¬vacia(p) hacer
        nodo ← desencolar(p)
        si valido(nodo) entonces dev nodo
        si no
            para cada hijo en compleciones(nodo) hacer
                si condiciones-poda(hijo) entonces encolar(hijo,p) fsi
            fpara
        fsi
    fmientras
ffun
```

USO: El recorrido en anchura se usará para (resto de casos recorrido en profundidad:

- Búsqueda del camino más corto (con menos nodos o aristas).
- En caso de grafos muy grandes o infinitos para hacer una exploración parcial.
- Cuando los nodos finales están cerca de la raíz.

Coste: Θ(max(a,n)) donde a=nº aristas y n=nº de nodos.

9.6 – VUELTA ATRÁS

9.6.0 – Esquema general: La vuelta atrás no es otra cosa que un recorrido en profundidad.

```
fun vuelta-atrás (ensayo)
    si válido (ensayo) entonces
        dev ensayo
    si no para cada hijo ϵ compleciones(ensayo)
        si condiciones-de-poda(hijo) hacer
            vuelta-atrás(hijo)
        fsi
    fsi
ffun
```

Habrá que especificar: ensayo, válido(si es solución), compleciones, condiciones-de-poda(es_prometedora).

Otros dos esquemas de este mismo algoritmo que encontramos a lo largo de los libros y ejercicios son:

```
fun vuelta-atrás (e: ensayo) dev (boolean,ensayo)
    si valido(e) entonces
        devolver (cierto,e)
    si no
        listaensayos ← compleciones (e)
        mientas no vacia (listaensayos) y no resultado hacer
            hijo ← primero(listaensayos)
            listaensayos←resto(listaensayos)
            si condiciones-de-poda(hijo) entonces
                (es_solucion,solución)o←vuelta-atras(hijo)
            fsi
        fmientras
        dev (es_solucion,solucion)
    fsi
ffun
```

Y el último esquema que encontramos:

```
fun vuelta-atrás (V[1..k]: nTupla)
    si valido(e) entonces
        devolver V
    si no
        para cada vector w(k+1)-prometedor hacer
            si condiciones-de-poda(w) hacer
                vuelta-atrás(w[1..k+1])
            fsi
        fpara
    fsi
ffun
```

9.6.1 – Problema de la mochila:

```
fun mochilava(i,r)
    b←0
    para k←i hasta n hacer
        si w[k]≤r entonces
            b←max(b,v[k]+mochilava(k,r-w[k]))
        fsi
    fpara
ffun
```

9.6.2 – Problema de las ocho reinas:

```
programa reinas2
    sol← permutación-inicial
    mientas sol≠permutacion-final y no solución(sol) hacer
        sol←permutacion-siguiente
    fmientras
    si solución(sol) entonces escribir sol
    si no escribir('No hay solucion') fsi
fprograma
```

9.7 – RAMIFICACIÓN Y PODA

9.7.1 – Cota se usa como condición adicional de poda:

```
fun ramificación-y-poda-en-anchura(ensayo)
        p←cola-vacia
        c←coste-minimo
        solución←solucion-vacia
        encolar(ensayo,p)
        mientras ¬vacia(p) hacer
                nodo←desencolar(p)
                si valido(nodo) entonces hacer
                        si coste(nodo) < c entonces hacer
                                solución ←nodo
                                c←coste(nodo)
                        fsi
                si no
                        para cada hijo en compleciones(nodo) hacer
                                si condiciones-poda(hijo) y cota(hijo)<c   entonces
                                        encolar(hijo,p)
                                fsi
                        fpara
                fsi
        fmientras
ffun
```

9.7.2 – Función de cota se usa para determinar orden de visitar los nodos y como condición de poda adicional (en este caso no podemos usar ni pilas ni colas).

```
fun ramificacion-y-poda(ensayo)
    m←monticulo-vacio
    cota-superior←inicializar-cota-superior
    solución ← solución-vacia
    añadir-nodo(ensayo,m)
    mientras ¬vacio(m) hacer
        nodo ←extraer-raiz(m)
        si valido(nodo) entonces hacer
            si coste(nodo)<cota-superior entonces
                solución←nodo
                cota-superior←coste(nodo)
            fsi
        si no
            si cota-inferior(nodo) ≥ cota-superior entonces   {(*)}
                devolver solución
            si no
                para cada hijo en compleciones(nodo) hacer
                    si condiciones-poda(hijo) y
                    cota-inferior(hijo)<cota superior entonces
                        añadir(hijo,m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

(*) – Si para un nodo cota-inferior≥cota-superior, no podremos llegar por el a otro nodo del montículo que sea una solución mejor, por lo que dejamos de explorar el resto del grafo implícito y devolvemos la mejor solución saliendo así del bucle.

Se organiza por montículo ya que así la raíz del montículo será aquel nodo con mejor "función de cota", es decir, tenemos en la raíz el nodo más prometedor.

> *cota-inferior*(nodo) : Suma de los costes de las tareas ya asignadas mas los costes mínimos de las no asignadas.

Habrá que especificar: compleciones, cota-inferior, valido y condiciones-de-poda (si la hay).

Coste: de normal en este tipo de problemas solo seremos capaces de obtener una cota superior.

Ejemplo de posible función de compleciones:

```
fun compleciones(ensayo)
        lista ← lista_vacia;
        si condiciones_de_poda(ensayo) entonces
                para cada rama válida hacer
                        w←generar_ensayo(rama)
                        lista← insertar(lista,w)
                fpara
        fsi
ffun
```