Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Tema 3: Estrategias Algorítmicas sobre estructura de datos no lineales

Resumen

Autor: Alberto Fernández Merchán
Asignatura: Algorímica y Modelos de Computación

Diciembre 2021

1. Algoritmos divide y vencerás

1.1. Introducción

Esta técnica consiste en descomponer un problema en un conjunto de subproblemas más pequeños del mismo tipo. Después se resuelven estos subproblemas y se combinan las soluciones.

Normalmente, su coste computacional se determina resolviendo relaciones de recurrencia.

1.2. Método General

1.2.1. Aproximación divide y vencerás

DIVIDIR: Se divide el problema original en varios subproblemas.

RESOLVER: Se resuelven esos subproblemas de forma recursiva o de forma directa si son suficientemente pequeños.

COMBINAR: Se combinan las soluciones para darle solución al problema original.

El esquema será eficiente cuando los subproblemas tengan una talla lo más parecida posible.
Para aplicar la técnica de divide y vencerás se deben cumplir los siguientes requisitos:

- Es necesario un método directo para resolver los problemas de tamaño pequeño.
- El problema original debe poder dividirse de forma sencilla en subproblemas.
- La solución debe obtenerse independientemente de los otros subproblemas.
- Es necesario un método **combinar** los resultados de los subproblemas.

1.2.2. Esquema general

```
función DivideVencerás (p: problema)
    Dividir (p, p1, p2, ..., pk) // pi son subproblemas de p
    para i = 1, 2, ..., k
        si = Resolver (pi)
    fpara
    return Combinar (s1, s2, ..., sk)
ffunción
```

Para resolver los subproblemas se utilizan llamadas recursivas del mismo algoritmo.

1.2.3. Esquema recursivo

```
función DivideVencerás (P: problema) return Solución
    si P suficientemente pequeño entonces
        return Solución = SoluciónDirecta(p)          // caso base
    sino
        Dividir (P, p1, p2, ..., pk) // descomponer  P en k subproblemas
        para i = 1, 2, ..., k
            si = DivideVencerás (pi)            // Resolver (pi)
        return  Solución = Combinar (s1, s2, ..., sk)
    fsi
ffunción
```

- **Pequeño**: Determina el caso base del problema.
- **SolucionDirecta**: Es el método que se usa para resolver el caso base.
- **Dividir**: Función que descompone el problema en subproblemas.
- **Combinar**: Combina los resultados de los subproblemas para obtener el resultado final.

1.3. Análisis de tiempos de ejecución

- **Dividir**: Divide el problema original de talla n en a subproblemas de tamaño n/b. Tiene coste lineal D(n).
- **Resolver**: Soluciona los a subproblemas de tamaño n/b. Tiene coste $aT(n/b)$.
- **Combinar**: Combina los resultados de los a subproblemas. Tiene coste lineal C(n).

En general, para las ecuaciones del tipo $T(n) = aT(n/b)+O(n^k \log^p(n))$, el teorema maestro se generaliza de forma:

$$T(n) \in \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \cdot \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \cdot \log^p n) & \text{si } a < b^k \end{cases}$$

1.4. Ejemplos de aplicación

1.4.1. MergeSort

- **Dividir**: Divide la secuencia de n elementos en dos subsecuencias de tamaño mitad.
- **Resolver**: Ordena las dos subsecuencias recursivamente utilizando MergeSort.
- **Combinar**: Combina las dos subsecuencias para generar la solución.
- **Caso Base**: Finaliza cuando la secuencia a ordenar contiene un único elemento.

```
MergeSort (A, p, r) /* Ordena un vector A desde p hasta r */
    if p<r {
      /* dividir en dos trozos de tamaño igual (o lo más parecido posible), es decir ⌈n/2⌉ y ⌊n/2⌋ */
       q = ⌊(p+r)/2⌋ ;              /* Divide */
      /* Resolver recursivamente los subproblemas */
       MergeSort (A,p,q);          /* Resuelve */
       MergeSort (A,q+1,r) ;       /* Resuelve */
       /* Combinar: mezcla dos listas ordenadas en O(n) */
        Merge (A,p,q,r);           /* Combina*/
    }
```

Análisis

- **Dividir**: Calcula el índice medio $O(1)$.
- **Resolver**: Resuelve recursivamente los subproblemas. $2T(n/2)$
- **Combinar**: Combina un vector de n elementos $O(n)$.

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1, \\ 2T(n/2) + c_2 \cdot n + c_3 & \text{si } n > 1, \end{cases}$$

De esta forma, resolviendo la relación de recurrencia sale que el coste computacional de este algoritmo (en el mejor caso) es de $O(n \log n)$.

Si hacemos que los subproblemas queden de una forma no equilibrada (caso peor), el coste temporal del algoritmo se convierte en $O(n^2)$.

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1, \\ T(1) + T(n-1) + c_2 \cdot n + c_3 & \text{si } n > 1, \end{cases}$$

Para mejorar el algoritmo es mejor no hacer llamadas recvursivas a mergesort cuando la talla es pequeña. En lugar de eso, será mejor llamar a algún algoritmo de ordenación como el algoritmo de inserción o de selección.

1.4.2. QuickSort

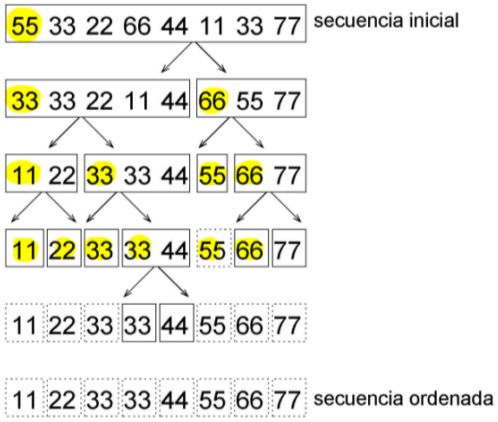
También se basa en la estrategia divide y vencerás. Sus pasos recursivos son:

- **Dividir**: El vector se reorganiza usando un procedimiento *Partition* en dos subvectores de forma que los elementos del subvector izquierdo son menores o iguales que los del derecho. El pivote también se calcula en el procedimiento de partición.

- **Resolver**: Los dos subvectores se ordenan resursivamente utilizando QuickSort.

- **Combinar**: No hay que hacer nada para combinar las soluciones ya que el vector completo ya queda ordenado.

- **Caso Base**: El algoritmo finaliza cuando la secuencia a ordenar contiene un único elemento (ya ordenado).



(En amarillo se ha señalado el elemento pivote de cada vector.)
En AMC usaremos el primer elemento de cada vector como pivote.

Análisis Siendo n el número de elementos a ordenar, el coste de Partition será de O(n) y reorganiza el vector en dos subvectores de tamaño i y (n-i). La recurrencia será:

$$T(n) = \begin{cases} k & \text{si } n \leq 1, \\ T(i) + T(n-i) + k' \cdot n & \text{si } n > 1, \end{cases}$$

En el **caso mejor**, los vectores estarán divididos en el mismo número de elementos. En este caso el sistema recurrente será el mismo que en el MergeSort.
En el **peor caso**, la división estaría desequilibrada, como en el peor caso del MergeSort.

Como es parecido al MergeSort, el coste computacional será de $O(n^2)$ en el peor caso, $O(n \log n)$ en el mejor y $O(n \log n)$ en el caso medio.

Para mejorar el coste temporal de este algoritmo se podría evitar, como en el mergesort, llamar recursivamente cuando la talla del vector es pequeña.

También podría elegirse el pivote como un elemento aleatorio en cada llamada recursiva.

1.4.3. Búsqueda del k-ésimo menor elemento

La solución directa sería ordenar el vector y acceder al elemento k-ésimo de la lista, pero eso conllevaría un coste de $n \cdot \log n$.
Si aplicamos la técnica divide y vencerás utilizaremos el método de *Partition*:

- **Dividir (Partition)** El vector se reorganiza en dos subvectores como en el quicksort.
- **Resolver**: Buscamos en el subvector correspondiente haciendo llamadas recursivas.
- **Combinar**: Comparamos la clave con el pivote y descartamos el subvector en el que no estará.
- **Caso Base**: La recursión finalzia cuando el subvector solo tiene un elemento.

En el mejor caso, el elemento a buscar es el menor o el mayor elemento del vector. De esta forma actúa como pivote y solo se particiona una vez el vector. $O(n)$.

En el peor caso, el vector estará ordenado de forma no decreciente y el elemento clave será el mayor. De esta forma el coste sería de $O(n^2)$.

En el caso medio asumimos que el vector se particiona por la mitad siempre, por lo que el coste será de $O(n)$.

2. Algoritmos Voraces

2.1. Introducción

Es uno de los esquemas más simples y utilizado. Se emplea para resolver problemas de optimización.

2.2. Esquema

2.2.1. Funcionamiento

El algoritmo voraz funciona por pasos:

1. Iniciamos con una solución vacía
2. En cada paso se escoge el siguiente elemento para añadir a la solución.
   Las decisiones no se pueden deshacer.
3. Acaba cuando el conjunto de elementos seleccionados sea la solución.

2.2.2. Elementos de los que consta la técnica

Se puede generalizar el proceso a un esquema algorítmico. En cada paso tenemos los siguientes conjuntos:

- Conjunto de candidatos pendientes.
- Conjunto de candidatos seleccionados.
- Conjunto de candidatos rechazados.

2.2.3. Ejemplo: Cambio de moneda

Añadir una moneda nueva a la solución hasta que el valor llegue a la cantidad P.

- Candidatos Iniciales: Todos los tipos de moneda disponibles.
- Conjunto Solución: Conjunto de monedas que suman P.
- Solución de la forma n-tupla donde cada x_i es el número de monedas tipo i.

```
funcion Devolver-cambio (int P): conjunto de monedas(X)
  const C={1,2,5,10,20,50,100,200} // C=monedas disponibles; conj. candidatos
  int X[1..N] ;                    // X= conjunto que contendrá la solución
  actual = 0                       //suma acumulada de la cantidad procesada
  para i = 1 hasta N
     X[i] = 0                      // inicialización(X)
  fpara
  mientras actual ≠ P              // no solución(X)
    j = el mayor elemento de C tal que C[j] ≤ (P-actual) // seleccionar(C)
    si j=0 entonces          // Si no existe ese elemento => no factible(j)
       devolver "No existe solución";
    fsi
    X[j] = (P-actual) div C[j]     // insertar(C,X)
    actual = actual + C[j]*X[j]
  fmientras
  devolver X                       // objetivo(X)
ffuncion
```

☐ **Orden de complejidad** del algoritmo: podemos encontrar el siguiente elemento en un tiempo constante (ordenando las monedas): **O(n)**.

2.3. Análisis de tiempos de ejecución

El orden de complejidad dependerá de muchos factores como: el número de candidatos (n), las funciones básicas a utilizar o el número de elementos de la solución (m).

Habrá que repetir como máximo n veces y como mínimo m veces.

- Comprobar si el valor actual es solución: $O(1)$ o $O(m)$.
- Selección de un elemento de los candidatos: $O(1)$ o $O(n)$.
- Comprobar si la solución es factible: $h(n)$.
- Insertar un nuevo elemento a la solución $j(n, m)$

Generalmente, el tiempo de ejecución es polinómico de la forma:

$$O(n * (f(m) + g(n) + h(m)) + m * j(n, m))$$

2.4. Ejemplos de aplicación

2.4.1. Selección de Actividades

Consiste en seleccionar un subconjunto de las actividades propuestas de manera que sean compatibles entre sí y que sea un subconjunto cardinal máximo.

El criterio que usaremos será seleccionar la **actividad que termine más pronto**.

```
función Selector_Actividades (c(1..n),f(1..n); var S(1..m))
// return conjunto_de_actividades (S(1..m), m ≤ n)
// c(1..n) y f(1..n) ordenados según el criterio (i<j ⇒ f(i)≤f(j))
    S ← {1}
    z ← 1       // z representa  a la última actividad seleccionada
    para i desde 2 hasta n hacer
        si c(i) ≥ f(z) entonces
            S ← S ∪ {i}
            z ← i
        fsi
    fpara
    return S
ffunción
```

2.4.2. Problema de la mochila

Consiste en llenar una mochila con fragmentos de objetos, maximizando la suma de los beneficios y sin superar la capacidad máxima.

Elegiremos como **criterio de selección** el mayor fragmento que quepa en la mochila con el mayor cociente beneficio/peso.

```
función Mochila_Frag (b (1..n), p(1..n), M) return X
    Ordenar_artículos_según_criterio() // por ej b_i/p_i max
    peso ← 0
    i ← 1  // representa el número del artículo a tratar
    mientras i ≤ n and peso + p(i ) ≤ M hacer  mientras haya objetos y no haya solución
        x (i ) ← 1
        peso ← peso + p(i)
        i ← i + 1
    fmientras
    // si i<n ⇒ peso + p(i) > M y ∀k: 1..i - 1 es x (k) = 1
    si i ≤ n entonces  si no hay ningún objeto entero →
        x (i ) ← (M -peso)/p(i )
        para k desde i+1 hasta n hacer → Pone el resto de objetos...
            x (k ) ← 0
        fpara
    fsi
    return x (1..n)
ffunción
```

2.4.3. Algoritmos de grafos greedy

Árboles de recubrimiento mínimo Consiste en obtener un nuevo grafo que solo contenga las aristas imprescindibles para una optimización de las conexiones de todos los nodos.

Las características son:

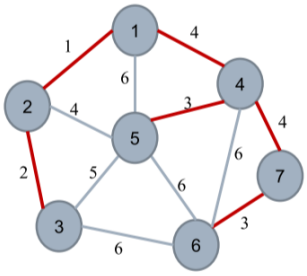
- Tiene $n-1$ arista.
- No se puede ni añadir ni eliminar una arista.
- Cualquier par de vértices está unido por un camino simple.

Algoritmo de Kruskal Partiendo del árbol vacío, se selecciona en cada paso la arista de menor coste que no provoque ciclo.

Los pasos son:

1. Empezar con un grafo sin aristas.
2. Seleccionar la arista con menor coste.
3. Si forma un ciclo, se elimina. Si no se añade a T.
4. Repetir 2 y 3 hasta tener n-1 aristas.

arista	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5) ...
peso	1	2	3	3	4	4	4	5



paso	selección	componentes conexas(conjuntos)						
inicial	-	1	2	3	4	5	6	7
1	(1,2)	1,2		3	4	5	6	7
2	(2,3)	1,2,3			4	5	6	7
3	(4,5)	1,2,3			4,5		6	7
4	(6,7)	1,2,3			4,5		6,7	
5	(1,4)	1,2,3,4,5					6,7	
6	(2,5) (ciclo-> rechazada)	1,2,3,4,5					6,7	
7	(4,7)	1,2,3,4,5,6,7						

Solución T = { (1,2), (2,3), (4,5), (6,7), (1,4) ,(4,7) }

```
función Kruskal ( G =(N,A) ) : árbol
    Ordenar A según longitudes crecientes;
    n := |N|;
    T := conjunto vacío;
    inicializar n conjuntos, cada uno con un nodo de N;
    /* bucle voraz */
    repetir
        a := (u,v) : arista más corta de A aún sin considerar;
        Conjunto U := Buscar (u);
        Conjunto V := Buscar (v);
        si Conjunto U <> Conjunto V entonces
            Fusionar (Conjunto U, Conjunto V);
            T := T U {a}
        fsi
    hasta |T| = n-1;
    devolver T
ffunción
```

Análisis Siendo n el número de vértices y m el número de aristas, se tiene que:

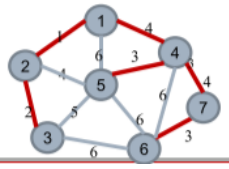
- Ordenar las aristas: $O(m \log m)$
- Inicializar los n conjuntos: $O(n)$
- 2*Buscar: $2 * (O(m))$
- Resto: $O(m)$

Por tanto, el coste del algoritmo de Kruskal será: $T(n) \in O(m \cdot \log n)$

Algoritmo de Prim Consiste en un único árbol que va creciendo hasta alcanzar todos los nodos.

Se selecciona la arista con menor coste que parte del nodo inicial. Se añade el vértice al conjunto de seleccionados y se elimina del conjunto de los no seleccionados para seguir añadiendo.

```
función Prim v.2 ( L[1..n,1..n] ) : árbol
  DistanciaMínima[1] := -1;
  T := conjunto vacío;
  para i := 2 hasta n hacer
      MásPróximo [i] := 1;
      DistanciaMínima [i] := L[i,1]
  fpara
  /* bucle voraz */
  repetir n-1 veces:
      min := infinito;
      para j := 2 hasta n hacer
          si 0 <= DistanciaMínima [j] < min entonces
             min := DistanciaMínima[j];
             k := j
          fsi
      fpara
      T := T U {(MásPróximo[k], k)};/*añadir arista a T
      DistanciaMínima [k] := -1;  /* añadir k a B */
      para j := 2 hasta n hacer
          si L[j,k] < DistanciaMínima[j] entonces
             DistanciaMínima [j] := L[j,k];
             MásPróximo[j] := k
          fsi
      fpara
  finrepetir
  devolver T
finfunción
```



L	1	2	3	4	5	6	7
1	∞	1	∞	4	6	∞	∞
2	1	∞	2	∞	4	∞	∞
3	∞	2	∞	∞	5	6	∞
4	4	∞	∞	∞	3	6	4
5	6	4	5	3	∞	6	∞
6	∞	∞	6	6	6	∞	3
7	∞	∞	∞	4	∞	3	∞

paso	T	MásPróximo K(min)	B (MásPróximo)	C	DistanciaMínima (vector distancias) 2	3	4	5	6	7
inicial	∅	∞	{1}	{2,3,4,5,6,7}	1	∞	4	6	∞	∞
1	{1,2}	2(1)	{1,2}	{3,4,5,6,7}	-1	2	4	4	∞	∞
2	{2,3}	3(2)	{1,2,3}	{4,5,6,7}	-1	-1	4	4	6	∞
3	{1,4}	4(4)	{1,2,3,4}	{5,6,7}	-1	-1	-1	3	6	4
4	{4,5}	5(3)	{1,2,3,4,5}	{6,7}	-1	-1	-1	-1	6	4
5	{4,7}	7(4)	{1,2,3,4,5,7}	{6}	-1	-1	-1	-1	3	-1
6	{7,6}	6(3)	{1,2,3,4,5,7,6}	∅	-1	-1	-1	-1	-1	-1

Análisis Siendo n el número de vértices y m el número de aristas, el coste computacional del algoritmo de Prim será:

- Inicialización: $O(n)$
- Bucle voraz: $O(n)$

Por tanto: $T(n) \in O(n^2)$

Comparación La comparación de ambos algoritmos es:

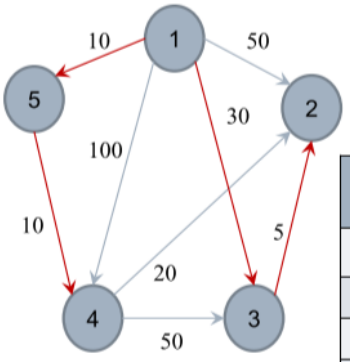
	Prim $\Theta(n^2)$	Kruskal $\Theta(mlogn)$
Grafo denso(cuando tiene muchas aristas): $m \to n(n-1)/2$	$\Theta(n^2)$	$\Theta(n^2logn)$
Grafo disperso(muy pocas aristas): $m \to n$	$\Theta(n^2)$	$\Theta(nlogn)$

2.4.4. Problema del camino mínimo

Algoritmo de Dijkstra Encuentra los caminos mínimos entre un nodo de origen y el resto de nodos del grafo.

```
función Dijkstra ( L[1..n,1..n] ) : vector [2..n]
     vector  D[2..n];
     vector  S[1..n];
     /* inicialización */
     C := {2, 3, ..., n};
     S := {1};
     para i := 2 hasta n hacer D[i] := L[1,i]  fpara
     /* bucle voraz */
     repetir n-2 veces:
          v := nodo de C que minimiza D[v];
          C := C-{v}; S :=S ∪ {v};
          para cada w en C hacer
            D[w] :=min(D[w], D[v]+L[v,w]) /*si D[w]> D[v]+L[v,w] ⇒ D[w]:=D[v]+L[v,w] */
          fpara
     frepetir
     devolver D
ffunción
```

3. Algoritmo de Dijkstra. Ejemplo.



L	1	2	3	4	5
1	∞	50	30	100	10
2	∞	∞	∞	∞	∞
3	∞	5	∞	∞	∞
4	∞	20	50	∞	∞
5	∞	∞	∞	10	∞

paso	V	C	D (vector distancias)			
			2	3	4	5
inicial	-	{2,3,4,5}	50	30	100	**10**
1	5	{2,3,4}	50	30	**20**	10
2	4	{2,3}	40	**30**	20	10
3	3	{2}	**35**	30	20	10

Tabla: Evolución del conjunto C y de los caminos mínimos.

➢ **Observación**: 3 iteraciones = n − 2

```
D[w] :=min(D[w], D[v]+L[v,w]) /*si D[w]> D[v]+L[v,w] ⇒ D[w]:=D[v]+L[v,w] */
```

Solución D = {35,30,20,10} ≡ (distancia desde el nodo 1 a cada nodo del grafo)

Análisis : Siendo n el número de vértices y m el número de aristas. El coste temporal será:

- Inicialización: $O(n)$
- Seleccionar nodo v: $O(n^2)$

Por tanto, el coste computacional total del algoritmo de Dijkstra será de: $T(n) \in O(n^2)$

3. Programación Dinámica

3.1. Introducción

La idea es la misma que en divide y vencerás, pero aplicando una estrategia diferente.

En ambas técnicas se descompone el problema de forma recursiva y se obtiene la solución aplicando un razonamiento inductivo.

Sin embargo, en divide y vencerás se aplica directamente la fórmula recursiva mientras que en programación dinámica se resuelven primero los problemas más pequeños guardando los resultados en una tabla.

Es un método ascendente, es decir, se resuelven primero los problemas más pequeños y luego se van combinando las soluciones para resolver los problemas más grandes.

3.2. Método General

3.2.1. Funcionamiento

Es un método general de optimización de procesos por etapas. Resuelve problemas caracterizados como recursivos, pero evita que las llamadas recursivas se solapen, evitando, de esta forma, la repetición de cálculos innecesarios.

3.2.2. Algoritmo de Floyd

Calcula los caminos mínimos entre cualquier par de nodos de un grafo. Para que un algoritmo de programación dinámica obtenga la solución correcta debe cumplir el principio de optimalidad de Bellman: `La solución óptima de un problema se obtiene combinando las soluciones óptimas de subproblemas`.

Es decir, si el camino mínimo de A a B pasa por C, entonces los trozos del camino de A a C y de C a B son también mínimos.

3.2.3. Pasos para aplicar programación dinámica

1. Obtener una descomposición recurrente del problema.
   - Ecuación Recurrente
   - Casos Base
2. Definir la estrategia de aplicación de la fórmula.
   - Tabla usadas por el algoritmo
   - Orden y forma de rellenarlas
3. Especificar cómo se recompone la solución final.

3.3. Análisis de tiempos de ejecución

Se basa en el uso de tablas donde almacenar resultados parciales. El tiempo de ejecución será de la forma: `Tamaño de la tabla` · `Tiempo en rellenar cada elemento de la tabla`.

3.4. Ejemplos de aplicación

3.4.1. Problema de la mochila sin fraccionamiento

La estrategia voraz de este problema no proporciona la solución óptima. A diferencia del esquema voraz, la programación dinámica sí que proporciona la solución óptima del problema.

```
/* La función mochila1 devuelve el beneficio total */
función Mochila1(p,b:[1..n] de entero; M:entero) devuelve natural;
     devuelve V(n,M)
ffunción Mochila1
/* El algoritmo V rellena un valor de la tabla y lo devuelve */
algoritmo V (i,j: natural) devuelve natural;  /* devuelve el valor de V[i,j]*/
   /* Inicializar los casos base */
  para i:=1 hasta n hacer V[i,0]:=0 fpara;
  para j:=0 hasta M hacer V[0,j]:=0 fpara;
  /* resto de los casos V[i, j]:= max(V[i-1, j], b_i + V[i-1, j-p_i]) */
  para i:=1 hasta n hacer
    para j:=1 hasta M hacer
      si j<p[i]  entonces /*j-p_i es negativo,caso -∞, y el máximo será el otro término.*/
          V[i,j]:=V[i-1,j]
        sino
          si V[i-1,j] ≥ V[i-1,j-p[i]]+b[i] entonces
              V[i,j]:=V[i-1,j]
            sino
              V[i,j]:=V[i-1,j-p[i]]+b[i]
          fsi
      fsi
    fpara
  fpara
falgoritmoV
```

Para evitar la repetición de cálculos se almacenan las soluciones parciales en una tabla de tamaño nxM cuyo elemento (i,j) almacenará el beneficio de una solución óptima.

Paso 2. Ejemplo. n= 3, M= 6, p= (2, 3, 4), b= (1, 2, 5)

(V[i, j]:= max (V[i-1, j], V[i-1, j-p_i]+ b_i))

i \ j	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	**6**

- Orden de complejidad del algoritmo : Cada componente de la tabla V se calcula en tiempo constante, luego el coste de construcción de la tabla es O(nM).

AMC_Tema 3 24

Para reconstruir la solución se compara el último elemento con el anterior. Si es igual no se ha seleccionado ningún elemento. Si es distinto significa que se ha seleccionado el elemento y se accede a la posición $V[M-p_k]$

Paso 3. Recomponer la solución óptima (cont.). Algoritmo.

```
función Objetos(M:natural; V[0..n][0..M] de natural b,p[1..n] de natural)
      test(n,M)
ffunciónObjetos
algoritmo test(i,j: natural)
  variables x:[1..n] de {0,1}
     j:= M
     para i:= n hasta 1 (dec 1) hacer
          si V[i, j] == V[i-1, j] entonces
               x[i]:= 0
          sino      /* V[i, j] == V[i-1, j-p_i] + b_i */
               x[i]= 1
               j= j - p_i
          fsi
     fpara
ffunciónntest
```

V	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	**6**

Esta solución solo es válida para pesos pertenecientes a los números enteros.

Análisis El coste del algoritmo será:

- Construcción de la tabla: $O(m \cdot n)$
- Reagrupar solucón: $O(n)$

3.4.2. Problema del cambio de moneda

Utilizando programación dinámica obtenemos la solución óptima de este problema.

```
/* La función Cambio devuelve el número mínimo de monedas necesario */
función Cambio(c:[1..n] de entero; P:entero) devuelve natural;
     devuelve D(n,P)
ffunción Cambio
/* El algoritmo D rellena un valor de la tabla y lo devuelve */
algoritmo D(i,j: natural) devuelve natural;  /* devuelve el valor de D[i,j]*/
   /* Inicializar caso base */
  para j:=0 hasta P hacer D[0,j]:=0 fpara;
  /* resto de los casos D[i, j]:= min(D[i-1, j], 1 + D[i, j-c_i]) */
  para i:=1 hasta n hacer
    para j:=0 hasta P hacer
      si j<c[i]  entonces /*j-c_i es negativo,caso +∞, y el mínimo será el otro término.*/
          D[i,j]:=D[i-1,j]
        sino
          si D[i-1,j] ≤ D[i,j-c[i]]+1 entonces
              D[i,j]:=D[i-1,j]
            sino
              D[i,j]:=D[i,j-c[i]]+1
          fsi
      fsi
    fpara
  fpara
falgoritmoD
```

3) Cómo recomponer la solución a partir de la tabla (continuación)

- Implementación:

```
función Monedas(M:natural; D[1..n][0..P] de natural c[1..n] de natural)
       test(n,P)
ffunciónObjetos
algoritmo test(i,j: natural)
   x: array [1..n] de entero  /* x[i] = número de monedas usadas de tipo i */
   x:= (0, 0, ..., 0)
   i:= n → nº tipo moneda
   j:= P → Cantidad a Cambiar } DC
   mientras (i≠0) AND (j≠0) hacer
     si D[i, j] == D[i-1, j] entonces
        i:= i - 1
     sino
        x[i]:= x[i] + 1
        j:= j - c_i
     finsi
   finmientras
```

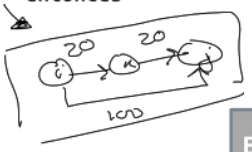
D	0	1	2	3	4	5	6	7	8
1 c_1=1	0	1	2	3	4	5	6	7	8
2 c_2=4	0	1	2	3	1	2	3	4	2
3 c_1=6	0	1	2	3	1	2	1	2	**2**

paso	i (tipo)	D[i,j]	D[i-1,j]	j (cantidad)	X (vector solución)		
					1	2	3
inicial	3	-	-	8	0	0	0
1	**3**	2	2	8	0	0	**0**
2	**2**	2	8	8-4=4	0	**1**	0
3	**2**	1	4	4-4=0	0	**2**	0

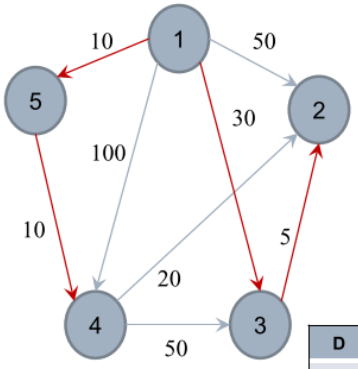
- **Ejemplo. n**= 3, **P**= 8, **c**= (1, 4, 6)
  - solución óptima es: $(x_1,x_2,x_3)=(0,2,0)$
  - con **2 monedas** de cantidad **4** (tipo 2)
- ¿Qué pasa si hay varias soluciones óptimas?

3.4.3. Problema del camino mínimo (Floyd)

```
función Floyd(g:grafo) devuelve D,C: vector[1..n,1..n] de natural
 variables D,C:vector[1..n,1..n] de natural; k,i,j:vértice;
 /* Inicializar los casos base de D, valor de la arista que une dos vértices, D0(i,j)=C(i,j) 1≤i≤n, 1≤j≤n; las
      diagonales se ponen a cero o bloquean */
 para i=1 hasta n hacer
    para j=1 hasta n hacer D[i,j]:=etiqueta(g,i,j);  /* ∞ si no hay arco */
    D[i,i]:=0;  /* ó D[i,i]:="-" */
 /* Inicializar C: C[i , j]= valor que representará el nodo predecesor a j en el camino mínimo desde i hasta j.
      Inicialmente se comienza con caminos de longitud 1, por lo que C[i , j]= i .*/
 para i=1 hasta n hacer
    para j=1 hasta n hacer C[i,j]:=i;  /* ∞ si no hay arco */
    C[i,i]:=0;  /* ó C[i,i]:="-" */
 /* resto de los casos Dk(i,j):=min(Dk-1(i,j), Dk-1(i,k) + Dk-1(k,j)) */
  para k=1 hasta n hacer
    para i=1 hasta n hacer
      para j=1 hasta n hacer
        si (i≠k) AND (j≠k) AND (i≠j) entonces
          si D[i,k]+D[k,j] < D[i,j]  entonces
            D[i,j]:=D[i,k]+D[k,j];
            C[i,j]:= C[k,j];
          fsi
        fsi
devuelve D,C;
Ffunción Floyd
{Post: D,C=caminosMínimos(g)}
```



Eficiencia temporal: $\Theta(n^3)$



1. Formar las **matrices iniciales *D*** y ***C***.
   - Inicialización de la matriz ***D***.
     - **D[i , j]**= valor que representa el **coste** de ir desde el nodo ***i*** al nodo ***j***
     - inicialmente en caso de no existir un arco entre ambos, el valor **D[i , j]**= ∞ caso de no existir un arco entre **el nodo *i* y el nodo *j***
   - Inicialización de la matriz ***C***.
     - **C[i , j]**= valor que representará el **nodo predecesor** a ***j*** en el camino mínimo desde ***i*** hasta ***j***.
     - Inicialmente se comienza con caminos de **longitud *1***, por lo que **C[i , j]= i** .

D	1	2	3	4	5
1	-	50	30	100	10
2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞
4	∞	20	50	-	∞
5	∞	∞	∞	10	-

C	1	2	3	4	5
1	-	1	1	1	1
2	∞	-	∞	∞	∞
3	∞	3	-	∞	∞
4	∞	4	4	-	∞
5	∞	∞	∞	5	-

Tablas: Inicialización de las matrices de costes D y de los caminos mínimos C.

4. Algoritmos de BackTracking

4.1. Introducción

Técnica para resolución de problemas de optimización, juegos y otros tipos. Consiste en un estudio exhaustivo y sistemático de todas las posibles soluciones. Por este motivo suele ser muy **ineficiente**.

Las etapas del algoritmo se pueden representar mediante un árbol de expansión. Cada nivel del árbol representa una etapa de la secuencia de decisiones. La solución se puede representar como una tupla.

Un estado puede ser **terminal** (solución) o **no terminal** (solución parcial). Un estado no terminal es factible cuando puede contener alguna solución factible.

El resultado es equivalente a realizar un recorrido en profundidad del árbol de soluciones. Estos árboles están implícitos en el algoritmo (no hace falta representarlos). Existen cuatro tipos de árboles de backtracking:

1. **Árboles Binarios**: Para problema de toma de decisiones sin importar el orden (Mochila 0/1 o encontrar un subconjunto que sume P).
2. **Árboles n-arios**: Para problemas donde se pueda elegir varias opciones para cada x_i. (Cambio de moneda o el problema de las n reinas).
3. **Árboles permutacionales**: Para problemas donde los x_i no se pueden repetir (Generar todas las permutaciones de 1..n o asignar n trabajos a n personas).
4. **Árboles combinatorios**: Para problemas iguales a los de los árboles binarios.

4.2. Esquema general

En cada momento el algoritmo debe encontrarse en un nivel K del árbol de soluciones (en ese nivel K se tiene una solución parcial). A continuación se comprueba si se puede añadir un nuevo elemento x_{k+1}:

En caso afirmativo se añade el elemento y se avanza al nivel $K + 1$.

En otro caso se prueban los hermanos del nodo visitado.

Si no hay más valores para x_k (ya se han visitado a todos los hermanos) se vuelve al nodo padre (nivel $k - 1$).
El algoritmo continua hasta que la solución parcial es completa o hasta que no queden más posibilidades.

```
procedimiento backtracking(var solucion[1..n])
  nivel:=1
  solucion:= solucion_INICIAL
  fin:=false
  repetir
    Generar(nivel,solucion)/*solucion[nivel]←generar(nivel,solucion)*/
    si EsSolucion(nivel,solucion) entonces
       fin:=true
    sino
       si Criterio(nivel, solucion) entonces
          nivel:=nivel + 1
       sino mientras not(HayMasHermanos(nivel,solucion)) hacer
             Retroceder(nivel,solucion)
          fmientras
       fsi
    fsi
  hasta fin=true
fprocedimiento
```

El esquema general puede variar dependiendo de las condiciones del problema:

- No es seguro que exista una solución.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) entonces
            fin:= true
        sino si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0)
                hacer Retroceder (nivel, s)
        finsi
    hasta fin OR (nivel==0)
```

Para poder generar todo el árbol de backtracking

- Se quiere almacenar todas las soluciones.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) entonces
            Almacenar (nivel, s)
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0)
                hacer Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

- En algunos problemas los nodos intermedios pueden ser soluciones
- O bien, retroceder después de encontrar una solución

- Problema de optimización.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    voa:= -∞; soa:= Ø
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) AND Valor(s) > voa entonces
            voa:= Valor(s); soa:= s
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT HayMasHermanos (nivel, s) AND (nivel>0)
                hacer Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

voa: valor óptimo actual
soa: solución óptima actual

4.3. Análisis de tiempos de ejecución

En general se obtienen órdenes de complejidad exponenciales y factoriales. Este dependerá del número de nodos generados y del tiempo requerido para cada nodo (constante).

Cada caso depende de como se realice la poda del árbol y del problema.

`Normalmente el tiempo de ejecución se puede obtener multiplicando el número de nodos del árbol por el tiempo de ejecución de cada nodo.`

4.4. Ejemplos de aplicación

4.4.1. Problema de la mochila sin fraccionamiento

- Se utiliza un árbol binario o combinatorio.
- Si $x_i = 0$: No se coge el objeto i.
- Si $x_i = 1$: Sí se coge el objeto i.
- Si $x_i = -1$: No se ha estudiado el objeto i.
- Las soluciones se encuentran en el nivel n. (En el combinatorio se encuentran en cualquier nivel)
- En el nivel i se estudia el objeto i.

```
Backtracking (var s: array [1..n] de entero)
    nivel:= 1;  s:= s_INICIAL
    voa:= -∞; soa:= Ø
    pact:= 0; bact:= 0 ..........> pact: Peso actual
                                   bact: Beneficio actual
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) AND (bact > voa) entonces
                voa:= bact; soa:= s
        si Criterio (nivel, s) entonces
                nivel:= nivel + 1
        sino
                mientras NOT HayMasHermanos (nivel, s) AND (nivel>0) hacer
                        Retroceder (nivel, s)
        finsi
    hasta nivel == 0
```

Para optimizar el algoritmo se puede utilizar un criterio de poda: Se puede establecer una **cota superior** para el problema de la mochila con fraccionamiento. (Sabemos que el algoritmo voraz obtiene la solución óptima de ese problema).

```
Criterio (nivel, s)
  si (pact > M) OR (nivel == n) entonces devolver FALSO
  sino
      bestimado:= bact + MochilaVoraz (nivel+1, n, M - pact)
      devolver bestimado > voa
  finsi
En el algoritmo principal:
  .....
  mientras (NOT HayMasHermanos (nivel, s) OR
            NOT Criterio (nivel, s)) AND (nivel > 0) hacer
    Retroceder (nivel, s)
```

Análisis El tiempo de ejecución, en el peor caso, será:
$T(n) = \sum_{i=1}^{n} 2^i \cdot (n - i + 1) = 2 \cdot 2^{n+1} - 2n - 4$

Mejoras Para mejorar el algoritmo se podría generar primero el 1 y luego el 0 en el árbol de decisiones. (Solo si la solución óptima es de la forma:
$s = \{1, 1, 1, X, X, 0, 0, 0\}$)

Si ordenamos los elementos por el cociente beneficio/peso, tendremos una solución de esa forma.

5. Ramificación y poda

5.1. Introducción

Se suele usar en problemas de optimización y en juegos. Es una mejora de la técnica de backtracking: realiza un recorrido sistemático del árbol de soluciones, pero no tiene por qué recorrerlo en profundidad y la estrategia de poda se realiza estimando cotas en cada nodo.

5.2. Método General

Para cada nodo i tendremos:

- CS(i): Cota superior.
- CI(i): Cota inferior.
- BE(i): Beneficio Estimado.

5.2.1. Maximización

Se podará un nodo i si se cumple que:

- $CS(i) \leq CI(j)$: Para algún nodo j generado.
- $CS(i) \leq Valor(s)$: Para algún nodo s solución final.

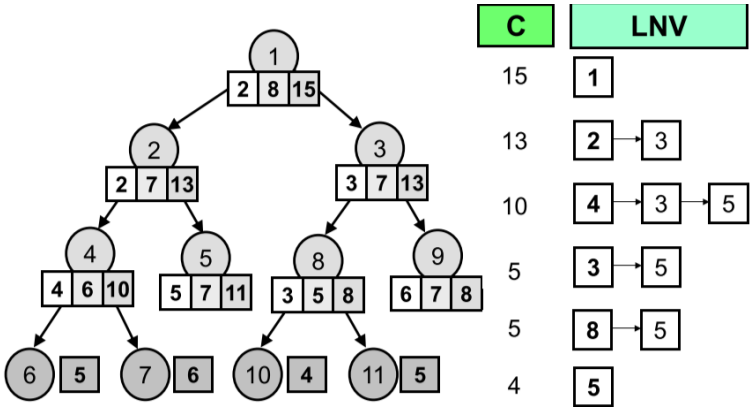
Existen diferentes tipos de recorridos utilizando una lista de nodos vivos (LNV) que contenga a los nodos pendientes de estudio.

Estrategia

- Sacar un elemento de LNV.
- Generar descendientes.
- Si no se podan, se meten en LNV.

Si LNV es una FIFO, entonces se hará un **recorrido en anchura**.
Si LNV es una LIFO, entonces se hará un **recorrido en profundidad**.
Estas estrategias realizan una búsqueda sin tener en cuenta los beneficios.

Si usamos la estrategia LC(Least Cost): Habrá que elegir el que tenga mayor beneficio para explorar a continuación.
En caso de empate habrá que utilizar una estrategia FIFO o LIFO.




```
RamificacionYPoda (raiz: Nodo; var s: Nodo)        // Minimización
    LNV:= {raiz}
    C:= CS(raiz)
    s:= ∅
    mientras LNV ≠ ∅ hacer
        x:= Seleccionar(LNV)                       // Estrategia de ramificación
        LNV:= LNV - {x}
        si CI(x) < C entonces                      // Estrategia de poda
            para cada y hijo de x hacer
                si Solución(y) AND (Valor(y)<Valor(s)) entonces
                    s:= y
                    C:= min (C, Valor(y))
                sino si NO Solución(y) AND (CI(y) < C) entonces
                    LNV:= LNV + {y}
                    C:= min (C, CS(y))
                finsi
            finpara
    finmientras
```

5.3. Análisis de tiempos de ejecución

El tiempo de ejecución dependerá del número de nodos recorridos y el tiempo usado en cada nodo.

En el caso promedio se obtienen mejores resultados que en backtracking.
En el caso peor se obtienen los mismos o peores tiempos que en backtracking.

5.4. Ejemplos de aplicación

5.4.1. Problema de la mochila sin fraccionamiento

```
Mochila01RyP (n: ent; b, p: array[1..n] de ent; var s: Nodo)
    LNV:= {raiz}
    C:= raiz.CI
    s:= ∅
    mientras LNV ≠ ∅ hacer
        x:= Seleccionar(LNV)        // Estrategia MB-LIFO
        LNV:= LNV - {x}
        si x.CS > C entonces                    // Estrategia de poda
            para cada y hijo de x hacer
                si Solución(y) AND (y.bact > s.bact) entonces
                    s:= y
                    C:= max (C, y.bact)
                sino si NO Solución(y) AND (y.CS > C) entonces
                    LNV:= LNV + {y}
                    C:= max (C, y.CI)
                finsi
            finpara
    finmientras
```

Para calcular:

- CI: El beneficio acumulado hasta el momento.
- CS: Solución de la mochila con fraccionamiento.
- BE: Usar el algoritmo voraz para el caso de la mochila sin fraccionamiento. Añadir objetos enteros por orden de b/p.

```
MochilaVoraz01 (a, b, Q): entero
   bacum:= 0
   pacum:= 0
   para i:= a hasta b hacer
      si pacum + p[i] ≤ Q entonces
         pacum:= pacum + p[i]
         bacum:= bacum + b[i]
      finsi
   finpara
   devolver bacum
```