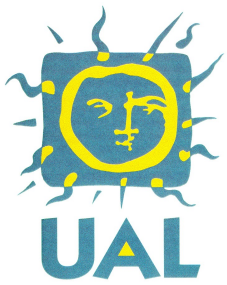
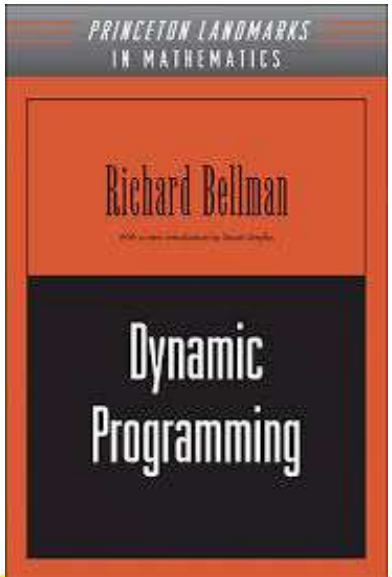
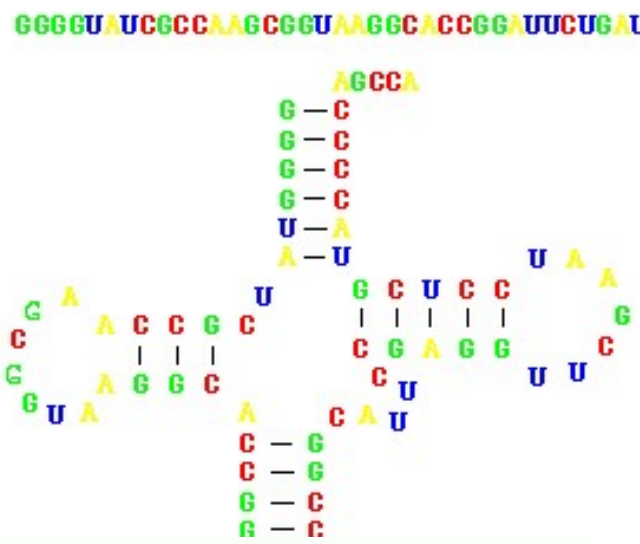


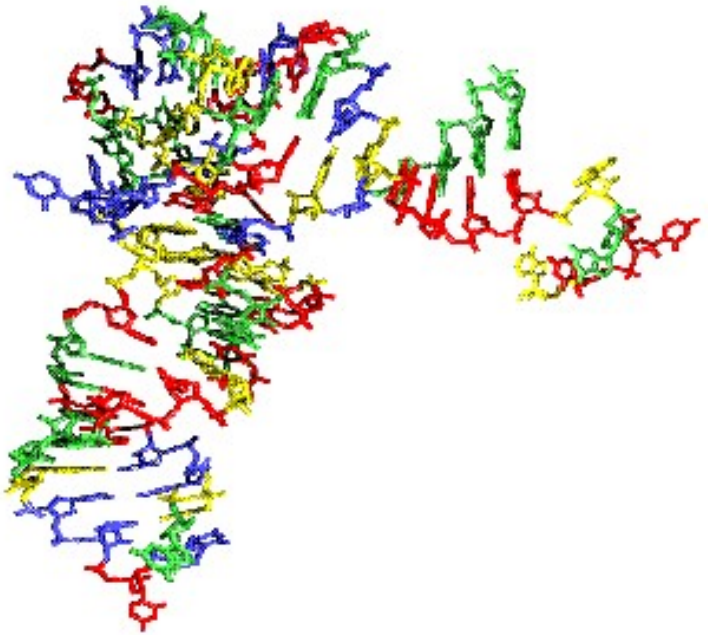
Tema 4 – Programación Dinámica.

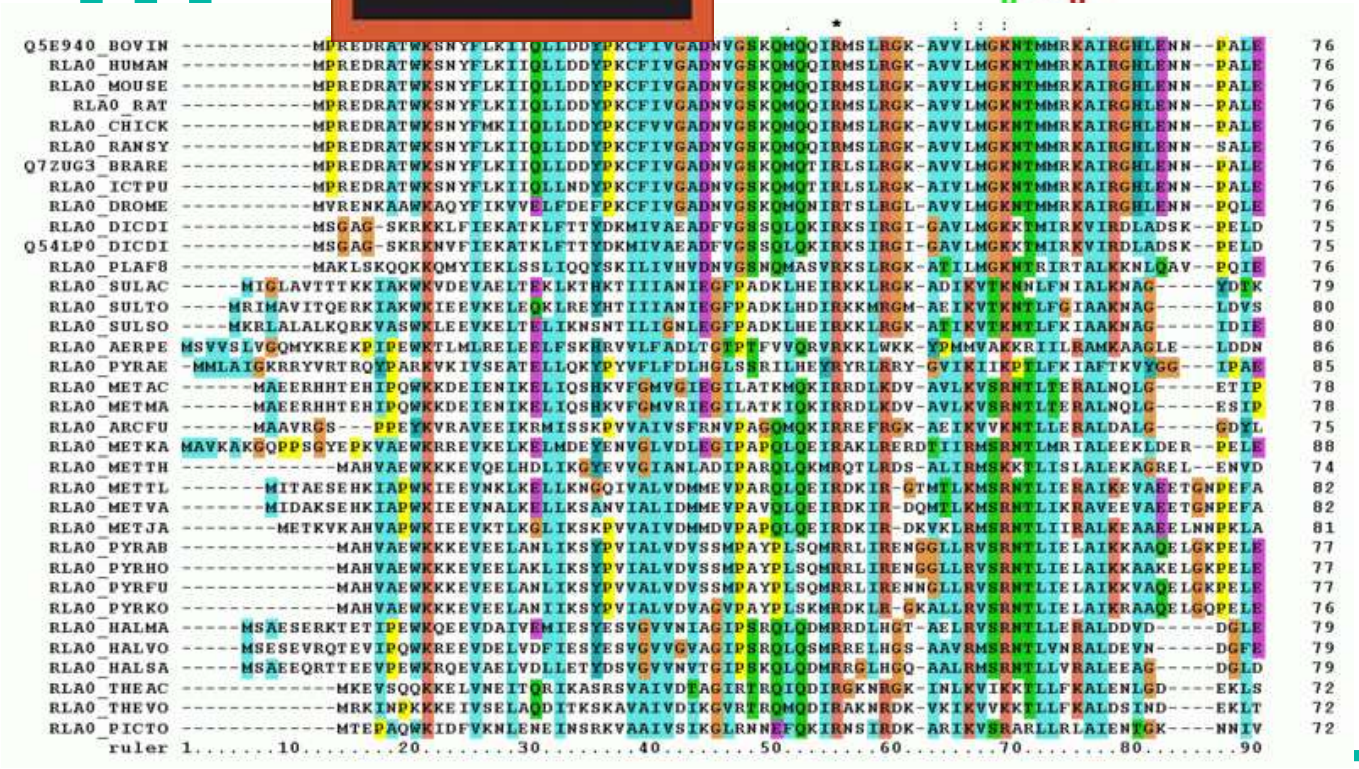
- *4.1. Introducción.*
- *4.2 Esquema general.*
- *4.3 Análisis de tiempos de ejecución.*
- *4.4 Ejemplos:*
  - *4.4.1 Caminos mínimos. Algoritmo de Floyd.*
  - *4.4.2 El problema de la Mochila 0/1.*
  - *4.4.3 El problema del Viajante.*
  - *4.4.4 Planificación de tareas.*
  - *4.4.5 Otros ejemplos.*

Programación Dinámica







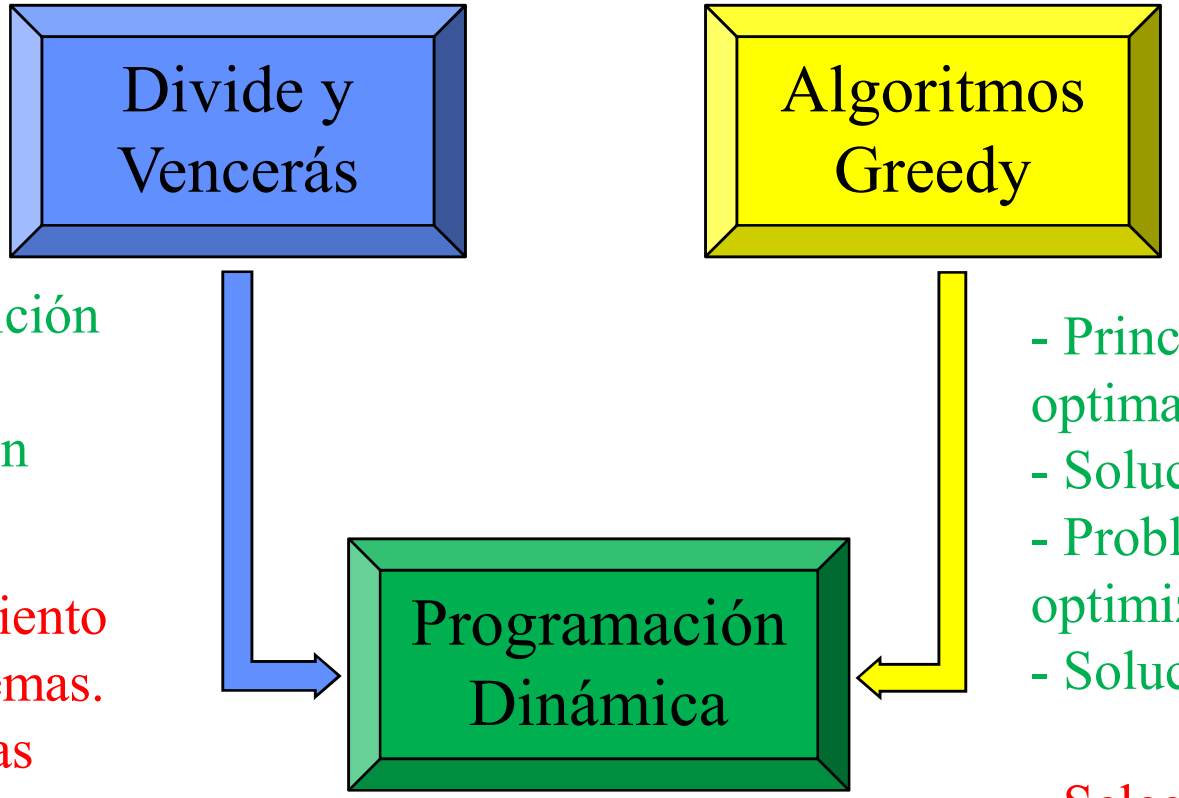


DNA
RNA
Protein

Sequencing

1 – Introducción.

Caracterización general:



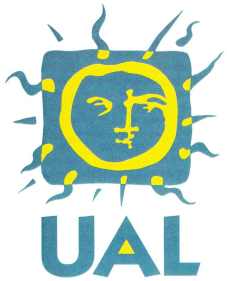
- Descomposición recursiva.
- Combinación soluciones.

- No solapamiento de sub-problemas. Sub-problemas independientes.
- Funciones recursivas.
- Método descendente.

Memorización de soluciones parciales.

- Principio de optimalidad.
- Solución iterativa.
- Problemas de optimización.
- Solución por etapas

- Selección voraz. Caso único.
- Incorporación inmediata a la solución.



1 – Introducción.

Caracterización, problemas no resueltos:

Divide y vencerás:

- ✓ Problemas con soluciones parciales no independientes.
- ✓ Problemas con coste recursivo muy alto.

Greedy:

- ✓ Problemas en que no se encuentra una función de selección voraz.
- ✓ Secuencias de decisiones con alternativas.

Programación dinámica

1 – Introducción.

Caracterización: razonamiento inductivo

Resolver el problema combinando las soluciones de problemas del mismo tipo, pero más pequeños. – como en divide y vencerás -

Similitudes:

- ✓Descomposición recursiva del problema.
- ✓Se aplica un razonamiento inductivo.
- ✓Se resuelve el problema original combinando las soluciones para subproblemas más pequeños.

1 – Introducción.

Caracterización: razonamiento inductivo

Resolver el problema combinando las soluciones de problemas del mismo tipo, pero más pequeños. – como en divide y vencerás -

Diferencias:

- Divide y vencerás: se aplica directamente la fórmula recursiva (algoritmo recursivo).

- Programación dinámica: se resuelven primero los problemas más pequeños, guardando los resultados en una tabla (algoritmo iterativo, no recursivo).

1 – Introducción.

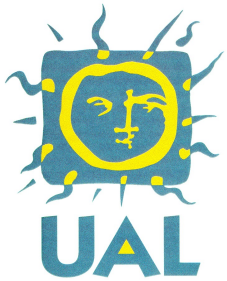
Caracterización: razonamiento inductivo

Resolver el problema combinando las soluciones de problemas del mismo tipo, pero más pequeños. – como en divide y vencerás -

Diferencias:

- Divide y vencerás: Método *descendente* (top-down), es decir, empezar con el problema original y descomponer recursivamente en problemas de menor tamaño. Partiendo del problema grande, descendemos hacia problemas más sencillos.

- Programación dinámica: Método *ascendente* (bottom-up), o sea, resolver primero los problemas pequeños, y después irlos combinando para resolver los de mayor tamaño. Partiendo de los problemas sencillos, avanzamos hacia los más grandes. Los problemas parciales se resuelven UNA SOLA VEZ, almacenando la solución (>eficiencia).



1 – Introducción.

Caracterización: secuencias de decisiones y optimización

Tipos de problemas que resuelve:

- ✓Problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).
- ✓Problemas mediante una secuencia de decisiones.

Similitud esquema voraz: mismo tipo de problemas que resuelve el esquema voraz.

Diferencias esquema voraz: se producen varias secuencias de decisiones y solamente al final se sabe cuál es la mejor de ellas.

1 – Introducción.

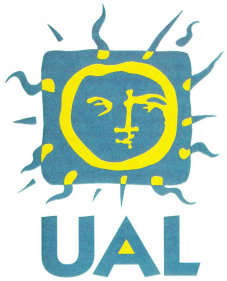
Caracterización del tipo de problemas que resuelven:

✓La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.

✓Dicha secuencia de decisiones ha de cumplir el principio de optimalidad.

✓Subproblemas optimales: La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor.

✓Solapamiento entre subproblemas: Al plantear la solución recursiva del problema, un mismo problema puede que se tenga que resolver más de una vez.



1 – Introducción.

Modo de trabajo:

- Enfoque *ascendente*: Primero se calculan las soluciones óptimas para problemas de tamaño pequeño. Luego, utilizándolas, se encuentran las de problemas de mayor tamaño.

- *Memorización* (clave): Almacenar las soluciones de los sub-problemas en alguna estructura de datos (tabla), para reutilizarlas posteriormente. La tabla se va llenando con las soluciones de los casos, empezando por los más pequeños y construyendo con ellos los grandes, hasta llegar al que se desea resolver. De esa forma, se consigue un algoritmo más eficiente que la fuerza bruta (o divide y vencerás), que resuelve el mismo sub-problema una y otra vez.

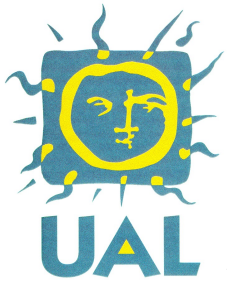
1 – Introducción.

Modo de trabajo:

Supongamos que un problema se resuelve tras tomar una secuencia d_1, d_2, …, d_n de decisiones.

Si hay d opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de d^n secuencias posibles de decisiones (explosión combinatoria).

La técnica de programación dinámica evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de tamaño creciente, y almacenamiento en una tabla de las soluciones óptimas de éstos, para facilitar la solución de los problemas más grandes.



1 – Introducción.

Modo de trabajo:

Evita calcular lo mismo varias veces:

- Cuando se calcula una solución, ésta se almacena.
- Antes de realizar una llamada recursiva para un sub-problema Q, se comprueba si la solución ha sido obtenida previamente:
  - Si no ha sido obtenida, se hace la llamada recursiva y, antes de devolver la solución, ésta se almacena.
  - Si ya ha sido previamente calculada, se recupera la solución directamente (no hace falta calcularla).
- Usualmente, se utiliza una matriz, que se rellena conforme son calculadas las soluciones a los subproblemas (espacio vs. tiempo). Suelen ser algoritmos *temporalmente eficientes* aunque con requerimientos de *memoria adicional* elevados.

1 – Introducción.

Resolución de problemas de optimización, comparación voraz:

La DP se utiliza para resolver *problemas de optimización* (minimizar o maximizar, bajo determinadas condiciones, el valor de una función), en los que se cumple el *principio de optimalidad* (forma menos estricta que para los voraces).

Podríamos hablar de un principio de optimalidad "relajado": La solución óptima de un problema es una combinación de soluciones óptimas de *algunos* de sus sub-casos.

La programación dinámica resuelve todos los sub-casos, lo que permite identificar los que conducen a la solución óptima. En los algoritmos voraces puede no ser posible identificar esos sub-casos, porque no exista la función de selección apropiada.

La programación dinámica puede llegar a resolver problemas sin solución óptima con el enfoque voraz, puesto que las decisiones no se toman a ciegas, sino considerando todos los sub-casos.

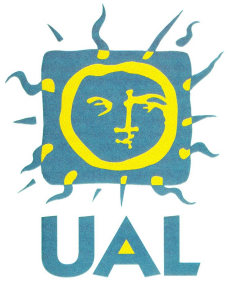
1 – Introducción.

Principio de optimalidad de Bellman (1957):

Principio de Optimalidad de Bellman [Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]:

"Una política óptima tiene la propiedad de que, sean cuales sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una solución óptima con respecto al estado resultante de la primera decisión".

"Cualquier sub-secuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al sub-problema que resuelve."



1 – Introducción.

Principio de optimalidad de Bellman (1957):

En Informática, un problema que puede descomponerse de esta forma, se dice que presenta subestructuras optimales (la base de los algoritmos greedy y de la programación dinámica).

El principio de optimalidad se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas.

Pero hay que tener *cuidado*: El principio de optimalidad *NO* nos dice que, si tenemos las soluciones óptimas de los subproblemas, entonces siempre podamos combinarlas para obtener la solución óptima del problema original…

1 – Introducción.

Principio de optimalidad de Bellman (1957): formulación

Dado un problema P con n elementos, si la secuencia óptima es:

$$e_1, e_2, ..., e_k, ..., e_n$$

entonces:

- ✓ $e_1, e_2, ..., e_k$ es solución al problema P considerando los k primeros elementos.
- ✓ $e_{k+1}, ..., e_n$ es solución al problema P considerando los elementos desde k+1 hasta n.

La solución óptima de cualquier instancia no trivial de un problema, es una combinación de las soluciones óptimas de sus sub-problemas.

Se busca la solución óptima a un problema como un proceso de decisión multi-etapa: se toma una decisión en cada paso, pero ésta depende de las soluciones a los sub-problemas que lo componen.

1 – Introducción.

Principio de optimalidad de Bellman (1957):

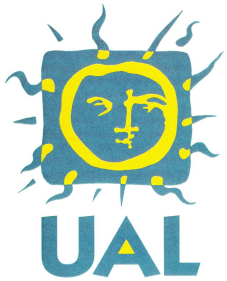
El principio de optimalidad se desarrolló como base de la Programación Dinámica (Bellman).

El enfoque del principio de optimalidad está inspirado en la teoría de control: Se obtiene la política óptima para un problema de control con n etapas, basándose en una política óptima para un problema similar, pero de n-1 etapas.

Origen del nombre de la "Dynamic Programming", problemas de matemática aplicada de los 50's:

"It's impossible to use dynamic in a pejorative sense"

"Something not even a Congressman could object to".



1 – Introducción.

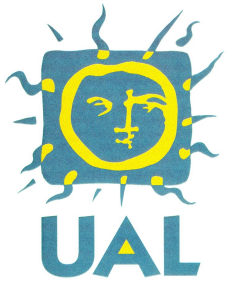
Principio de optimalidad de Bellman (1957): Ejemplo

Cambio en monedas (no algoritmo, aplicación principio optimalidad):

- ✓La solución óptima para 0.07 euros es (0.05 + 0.02) euros.
- ✓La solución óptima para 0.06 euros es (0.05 + 0.01) euros.
- ✓La solución óptima para 0.13 euros NO es la que se obtendría a partir de la descomposición 0.13 = 0.07 + 0.06, es decir, (0.05 + 0.02) + (0.05 + 0.01) euros.
- ✓Sin embargo, sí que existe alguna forma de descomponer 0.13 euros de modo que las soluciones óptimas a los subproblemas nos den una solución óptima:

$$0.13 = 0.11 + 0.02$$

nos llevaría a la solución (0.10 + 0.01) + 0.02 euros.



1 – Introducción.

Ejemplo de programación dinámica: sucesión de Fibonacci

Fib (n) = 0, para n = 0; Fib(1) = 1, para n =1.
Fib (n) = Fib (n-1) + Fib (n-2), para n > 1.

Esquema recursivo:

```
método fibRec(entero n): entero                    // n >=0
        si n ≤ 1 entonces
                retorna n
        si no
                retorna fibRec(n-1) + fibRec(n-2)
        fsi
fmétodo
```

Poco eficiente, orden exponencial.
Solapamiento de subproblemas…
No condiciones divide y vencerás…

1 – Introducción.

Ejemplo de programación dinámica: sucesión de Fibonacci

Solución con tiempo lineal, mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento, para poder reutilizarlos.

Fib (0)	Fib (1)	Fib (2)	⋯	Fib (n)

```
método fibIter(entero n): entero
        entero [0..n] fib
        si n ≤ 1 entonces
                retorna n
        si no
                fib[0] = 1
                fib[1] = 1
                para i = 2 hasta n hacer
                    fib[i] = fib[i-1] + fib[i-2]
                fpara
                retorna fib[n]
        fsi
fmétodo
```

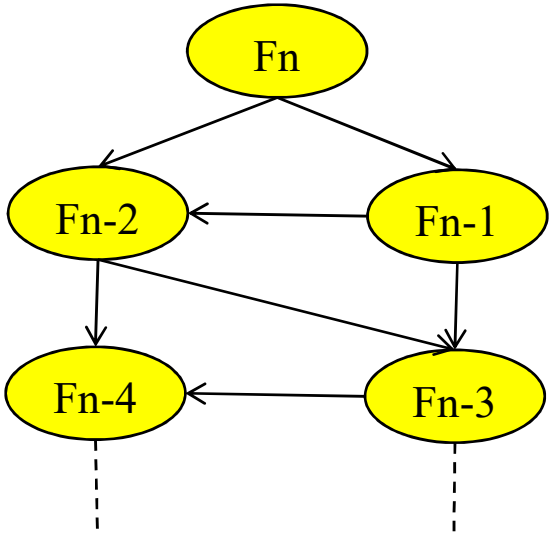
Algoritmo iterativo sucesión de Fibonacci utilizando la tabla.

O(n)

1 – Introducción.

Ejemplo de programación dinámica: sucesión de Fibonacci, mejora del algoritmo

Para la sucesión de Fibonacci, no hace falta almacenar todos los valores previos, basta con los dos últimos.

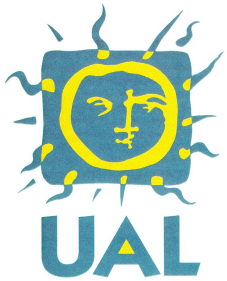



```
método fibIter2(entero n): entero
        entero previo, actual
        si n ≤ 1 entonces
                retorna n
        si no
                previo = 1
                actual = 1
                para i = 2 hasta n hacer
                        fib = previo + actual
                        previo = actual
                        actual = fib
                fpara
                retorna fib
        fsi
fmétodo
```

1 – Introducción.

Ejemplo de programación dinámica: sucesión de Fibonacci, comparación de eficiencias

	Eficiencia temporal	Eficiencia memoria
1. Recursiva:	$O(1.62^n)$	O(n) **
2. Dinámica 1:	O(n)	O(n)
3. Dinámica 2:	O(n)	O(1)



2 – Esquema general.

No hay un esquema algorítmico único. Hay una serie de fases:

1. Verificar que la solución puede alcanzarse a partir de una sucesión de decisiones y que ésta cumple el principio de optimalidad de Bellman, para lo cual hay que encontrar y caracterizar la estructura de una solución óptima.

2. Definir de forma recursiva la solución óptima del problema (en función de los valores de las soluciones para sub-problemas de menor tamaño). Es decir, obtener una descomposición recurrente del problema, compuesta de casos base y ecuación recurrente. Recordar: descomposición recursiva no necesariamente implica implementación recursiva.

Punto clave: obtener la descomposición recurrente. Requiere mucha creatividad...

2 – Esquema general.

No hay un esquema algorítmico único. Hay una serie de fases:

3.Calcular el valor de la solución óptima utilizando un enfoque ascendente:

- Se determina el conjunto de sub-problemas que hay que resolver (el tamaño de la tabla).
- Se identifican los sub-problemas con una solución trivial (casos base).
- Se utiliza la expresión recursiva para rellenar la tabla de soluciones parciales, hasta encontrar la solución óptima al problema planteado. Se van calculando los valores de soluciones más complejas, a partir de los valores previamente calculados. Habrá que definir la estrategia de aplicación de la fórmula: orden y forma de rellenar la tabla.

2 – Esquema general.

No hay un esquema algorítmico único. Hay una serie de fases:

4.Construir la solución óptima a partir de los datos almacenados en la tabla al obtener soluciones parciales, desandando sobre la tabla el camino que nos ha llevado a la solución óptima.

Elementos clave:

- Descomposición recursiva del problema.
- Construcción iterativa del conjunto de soluciones parciales, con un esquema botton-up.
- Almacenamiento de las soluciones parciales, son necesarias estructuras de datos que ocupan memoria (a veces mucha).
- La solución "final" a veces requiere de un post-procesamiento del resultado del método "dinámico".

2 – Esquema general.

Aplicación del razonamiento inductivo:

- ✓ ¿Cómo reducir un problema a subproblemas más simples?
- ✓ ¿Qué parámetros determinan el tamaño del problema (es decir, cuándo el problema es más simple)?
- ✓ ¿Cómo obtener la fórmula?

Idea: ver lo que ocurre al tomar una decisión concreta, es decir, interpretar el problema como un proceso de toma de decisiones.

Conclusión: la programación dinámica resuelve todos los subcasos, determinando los que son relevantes, para combinarlos en una solución óptima del caso original.

3 – Análisis de tiempos de ejecución.

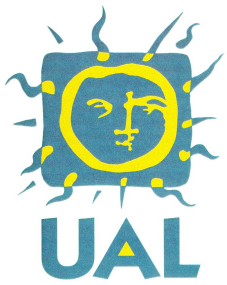
El *tiempo de ejecución* dependerá de las características concretas del problema que se vaya a resolver. En general, será de la forma:

Tamaño de la tabla x Tiempo de rellenar cada elemento de la tabla.

Un aspecto importante, ya que la programación dinámica se basa en el uso de tablas, donde se almacenan los resultados parciales, es que hay que tener en cuenta la *memoria* que puede llegar a ocupar la tabla.

Algunos de estos cálculos parciales pueden ser innecesarios.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica, consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico, conduzca a una complejidad espacial demasiado elevada.



4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Definición del problema:

Sea un grafo G = <N, A> donde N es el conjunto de nodos (vértices) y A el de aristas. Se trata de un grafo dirigido y ponderado (las aristas tienen pesos no negativos.)

Objetivo: resolver el problema de encontrar los caminos de coste mínimo entre cualquier par de vértices del grafo G.

Coste (peso o *longitud)* de un camino: suma de los pesos de las aristas que lo componen.

Utilidad: El grafo puede representar una distribución geográfica, donde las aristas dan el coste (precio, distancia...) de la conexión entre dos lugares, y se desea averiguar el camino más corto (barato...) para llegar a cualquier punto, partiendo de cualquier otro.

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Soluciones al problema:

- Por fuerza bruta (de orden exponencial). Se prueban todas las combinaciones posibles.
- Aplicar el algoritmo de Dijkstra (voraz) para cada vértice.
- Algoritmo de Floyd (programación dinámica).

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Aplicación del principio de optimalidad:

Si $e_1, e_2, \ldots, e_k, e_{k+1}, \ldots, e_n$ es un camino de coste mínimo de e_1 a e_n, entonces:

$e_1, e_2, \ldots, e_k$ es un camino de coste mínimo de e_1 a e_k

$e_k, e_{k+1}, \ldots, e_n$ es un camino de coste mínimo de e_k a e_n.

Aplicación del principio:

Si k es el vértice intermedio de mayor índice en el camino óptimo de i a j, entonces el subcamino de i a k es un camino óptimo de i a k que, además, sólo pasa por vértices de índice menor que k.

Con el subcamino de k a j ocurre lo análogo.

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Definición recursiva de la solución:

Razonamiento *inductivo*: para calcular los caminos mínimos pudiendo pasar por los k primeros vértices del grafo, usamos los caminos mínimos pasando por los k-1 primeros.

Sea $D_k(i, j)$ la longitud (o distancia) del camino más corto (de coste mínimo) de i a j, usando sólo los k primeros vértices del grafo como puntos intermedios, es decir, que no pasa por ningún vértice de índice mayor que k.

Expresión recursiva: $D_k(i, j) = \min \{ D_{k-1}(i, j), (D_{k-1}(i, k) + D_{k-1}(k, j)) \}$

Caso base: $D_0(i, j)$ = peso arista (i, j) [∞ si no existe la arista, 0 si i = j]

En ambas expresiones se cumple: $1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$.

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Definición recursiva de la solución:

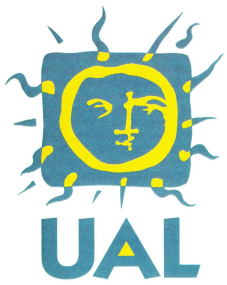
Un camino óptimo de i a j que no pase por ningún vértice de índice mayor que k, o bien pasa por el vértice k, o no pasa.

a) Si pasa por k, entonces:

$$D_k(i, j) = D_{k-1}(i, k) + D_{k-1}(k, j)$$

b) Si no pasa por k, entonces ningún vértice intermedio tiene índice superior a k-1:

$$D_k(i, j) = D_{k-1}(i, j)$$



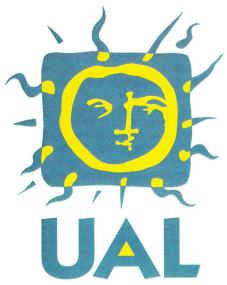
4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Solución con programación dinámica: algoritmo de Floyd

a) Se aplica un enfoque *inductivo*: calcular los caminos mínimos pasando por los k primeros vértices, con los caminos mínimos pasando por los k-1 primeros.

b) Es aplicable el principio de optimalidad: si en el camino mínimo de v_i a v_j, v_k es un vértice intermedio, los caminos de v_i a v_k y de v_k a v_j han de ser a su vez caminos mínimos.

c) Utiliza una matriz para evitar la repetición de los cálculos.



4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Datos de entrada y salida del algoritmo de Floyd:

Entradas:

g – Grafo ponderado y dirigido (nodos, aristas)

Salidas:

d - array [1..n] [1..n] de enteros, es la matriz de distancias entre cada par de vértices.

p - array [1..n] [1..n] de enteros, es la matriz de vértices intermedios en los caminos entre cada par de vértices, para poder reconstruir los caminos. Almacena el vértice intermedio de mayor índice.

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Esquema algoritmo de Floyd:

```
Pre: g es un grafo dirigido, ponderado con pesos no negativos.}
método floyd(grafo g): entero array [1..n] [1..n], entero array [1..n] [1..n]
  array [1..n] [1..n] de entero d, p        // d es matriz distancias par vértices
  entero i, j, k                            // p es matriz de vértices intermedios
  para i=1 hasta n hacer                    // Inicialización de d y p
      para j=1 hasta n hacer
          Inicializar d [i] [j]   // A valor nulo para i=j, al peso de la arista (i, j)
                                  // si existe, y a ∞ si no existe.
          Inicializar p[i] [j]    // A valor nulo para i=j y para cuando existe
                                  // la arista (i, j), ya que en ese caso no habrá
                                  // vértice intermedio, y a ∞ si no existe la arista.
      fpara
  fpara
  …..
```

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Esquema algoritmo de Floyd:

```
Pre: g es un grafo dirigido, ponderado con pesos no negativos.}
método floyd(grafo g): entero array [1..n] [1..n], entero array [1..n] [1..n]
  …..
  para k=1 hasta n hacer          // Se va analizando el efecto de cada nodo
     para i=1 hasta n hacer
          para j=1 hasta n hacer
               si d[i][k] + d[k][j] < d[i][j] entonces
                    d [i][j] = d[i][k] + d[k][j]
                    p [i][j]  = k
               fsi
          fpara
     fpara
  fpara
  retorna d, p
fmétodo
```

{Post: d contiene la distancia de los caminos mínimos entre cada par de vértices de g; p contiene el vértice intermedio en el camino mínimo entre cada par de vértices de g, para poder reconstruir los vértices por los que pasa dicho camino mínimo.}

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Eficiencia comparada:

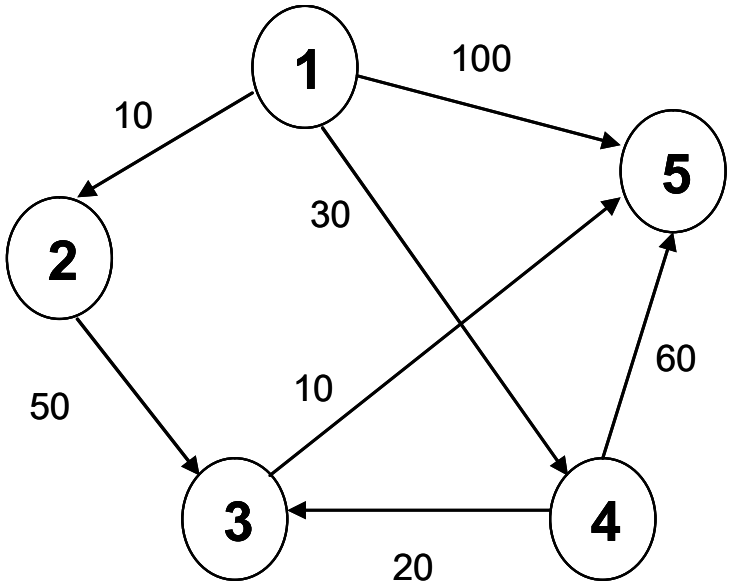
- Tiempo de ejecución: $O(n^3)$. El orden de complejidad temporal se debe al triple bucle anidado, en cuyo interior hay tan sólo operaciones constantes.

- Podemos resolver el problema aplicando n veces Dijkstra (algoritmo *voraz*), eligiendo cada vez un nodo distinto como origen. Usando para el grafo una representación con matriz de adyacencia, la complejidad es igual: $O(n \cdot n^2) = O(n^3)$, aunque el interior del bucle en Floyd es más simple, por tanto, la constante oculta es más pequeña (luego es más rápido).

- Espacio: Floyd exige $\Theta(n^2)$ mientras que Dijkstra precisa $\Theta(n)$.

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Ejemplo de aplicación del algoritmo de Floyd :



d	1	2	3	4	5
1					
2					
3					
4					
5					

p	1	2	3	4	5
1					
2					
3					
4					
5					

4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Recuperación de un camino (inicio, fin) de la tabla p:

```
método caminoFloyd(array[1..n][1..n] de entero p, entero origen, destino)
// Lanzador del algoritmo recursivo de recuperación del camino intermedio.
        entero k
        k = p[origen][destino]
        si k ≠ ∞ entonces
                Escribir en pantalla (origen)
                caminoIntermed(p, origen, destino)
                Escribir en pantalla (destino)
        si no
                Escribir en pantalla "No existe el camino"
        fsi
fmétodo
```

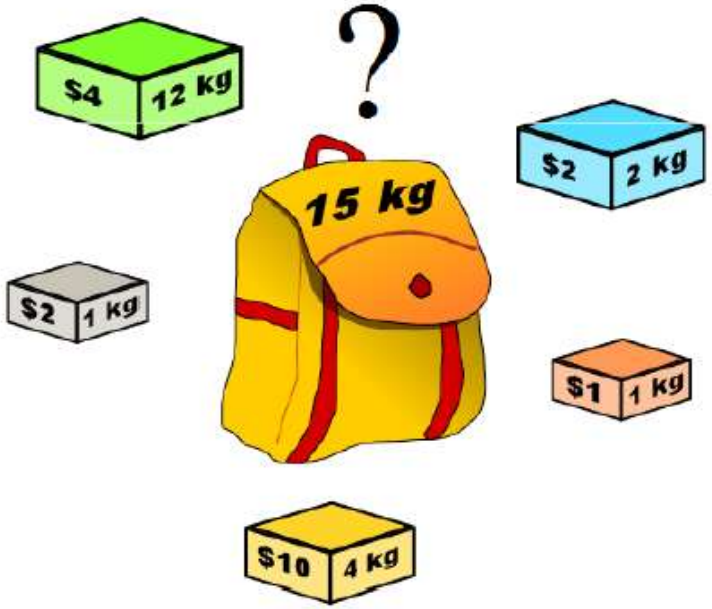
4 – Ejemplos: Caminos mínimos. Algoritmo de Floyd.

Recuperación de un camino (inicio, fin) de la tabla p:

```
método caminoIntermed(array[1..n][1..n]de entero p, entero origen, destino)
 // Algoritmo recursivo de recuperación del camino intermedio.
        entero k
        k = p [origen][destino]
        si k ≠ 0 entonces          // Si k=0 , arista y no hay camino intermedio.
                caminoIntermedio(p, origen, k)
                Escribir en pantalla (k)
                caminoIntermedio(p, k, destino)
        fsi
fmétodo
```

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento formal:



a) S es el conjunto de objetos
b) n es el número de objetos;
c) b_i ($b_i > 0$) es el beneficio (o valor) asociado al objeto i;
d) p_i ($p_i > 0$) es el peso del objeto i;
e) P es el peso máximo soportado por la mochila;
f) x_i es la presencia/ausencia del objeto i, su valor puede ser 0 o 1.

$$\max \sum_{1 \le i \le n} x_i b_i \quad sujeto\ a \sum_{1 \le i \le n} x_i p_i \le P \quad con\ x_i \in \{0,1\}$$

4 – Ejemplos: El problema de la Mochila 0/1

Estrategia voraz:

No genera el óptimo siempre.

Tomar el objeto con mayor beneficio por unidad de peso.

Ejemplo:

n=3 P=15

$(b_1, b_2, b_3) = (38, 40, 24)$

$(w_1, w_2, w_3) = (9, 6, 5)$

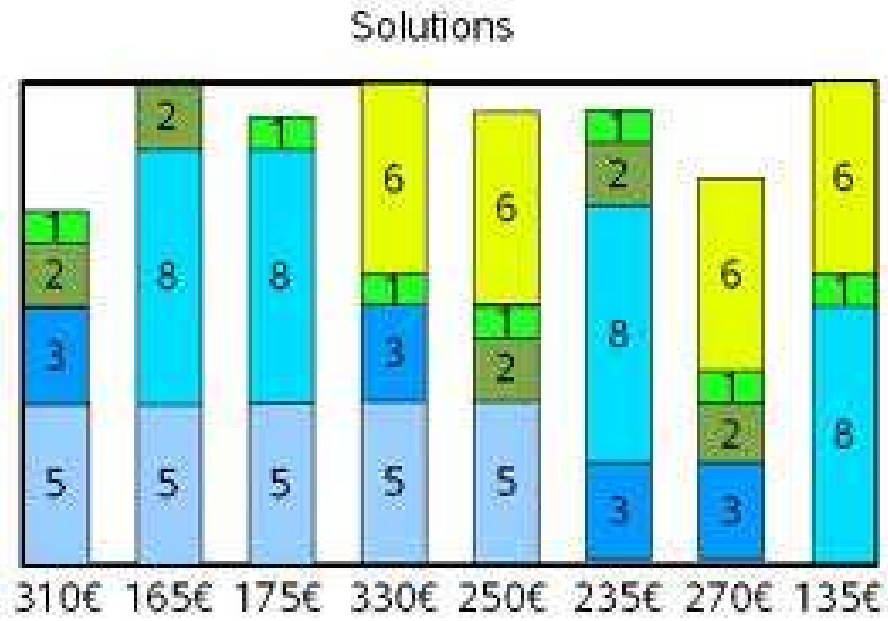
Solución voraz: $(x_1, x_2, x_3) = (0, 1, 1)$
Beneficio 64.

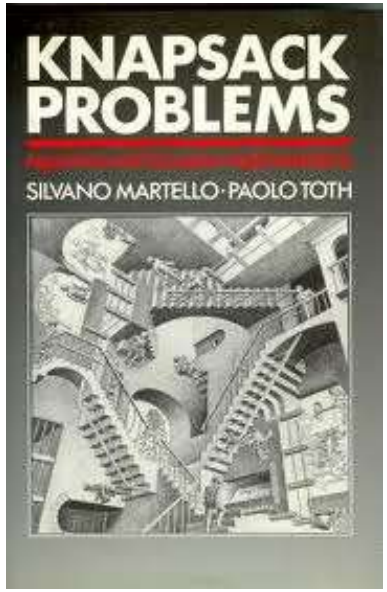
Solución óptima: $(x_1, x_2, x_3) = (1, 1, 0)$
Beneficio 78.

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

1. Secuencia de decisiones cumpla principio de optimalidad.
2. Formulación recursiva.
3. Buscar estructura de datos reutilización resultados recurrencias.





4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

1. Secuencia de decisiones cumpla principio de optimalidad.

Secuencia de decisiones (1..n) coger o no coger el objeto.

Resolver (1..n)	Coger n + Resolver (1..n-1)	Beneficio b_n	Peso $P-p_n$
	No coger n + Resolver (1..n-1)	Beneficio -	Peso P
Para un objeto k	Coger k + Resolver (1..k-1, $P-p_k$)	Beneficio b_k	Peso $P-p_k$
	No coger k + Resolver (1..k-1, P)	Beneficio -	Peso P

Un problema de tamaño k se resuelve con dos problemas de tamaño k-1, con distinto peso posible….

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

1. Secuencia de decisiones cumpla principio de optimalidad.

$S_k \equiv$ conjunto de los k primeros objetos (de 1 a k)

B(k, p) como el beneficio (la ganancia) de la mejor solución obtenida a partir de los elementos de S_k para una mochila de capacidad p.

B(n, P) es el beneficio del problema original. Óptimo.

$\{x_1, x_2, ..., x_n\}$ secuencia de decisiones que conducen a obtener el valor B(n, P) / $x_i \in \{0,1\}$ (no se guarda, se guarda)

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

1. Secuencia de decisiones cumpla principio de optimalidad.

Hay dos opciones distintas:

a)Que $x_n = 1$. La subsecuencia de decisiones $x_1, x_2, ..., x_{n-1}$ ha de ser también óptima para el problema $B(n-1, P-p_n)$, ya que si no lo fuera y existiera otra subsecuencia de decisiones $d_1, d_2, ..., d_{n-1}$ óptima, la secuencia $d_1, d_2, ..., d_{n-1}, x_n$ también sería óptima para el problema $B(n, P)$, lo que contradice la hipótesis.

b)Que $x_n = 0$. Entonces la subsecuencia de decisiones $x_1, x_2, ..., x_{n-1}$ ha de ser también óptima para el problema $B(n-1, P)$.

Principio de optimalidad

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

2. Formulación recursiva.

La mejor selección de elementos del conjunto S_k para una mochila de tamaño p, se puede definir en función de selecciones de elementos de S_{k-1} para mochilas de menor capacidad…

Definición recursiva de B(k, p):

$$B(k,p) \begin{cases} B(k-1,p) & \text{Si } x_k = 0 \\ B(k-1,p-p_k)+b_k & \text{Si } x_k = 1 \end{cases}$$

Solución, la que nos da el máximo:

$$B(k,p) = \max\{B(k-1,p),\ B(k-1,\ p-\ p_k)+b_k\}$$

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

2. Formulación recursiva.

Casos base:

- ✓Si p=0, no se pueden incluir objetos: B(k, 0) = 0.
- ✓Si k=0, tampoco se pueden incluir: B(0, p) = 0.
- ✓Si p ó k son negativos, el problema es irresoluble.

Caso de partida:

```
Método mochila1(entero [1..n] p,b; int P): entero // Beneficio obtenido
        // Utiliza los vectores p y b como globales para no complicar..
        retorna B(n, P)
fmétodo
```

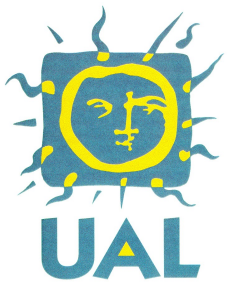
4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

2. Formulación recursiva.

```
Método B(entero j,w): entero    // Devuelve beneficio.
        si j=0 entonces
                retorna 0
        sino
                si w<p[j] entonces
                        retorna B(j-1,w)
                sino
                        si B(j-1,w)≥B(j-1,w-p[j])+b[j] entonces
                                retorna B(j-1,w)
                        sino
                                retorna B(j-1,w-p[j])+b[j]
                        fsi
                fsi
        fsi
fmétodo
```

NO es aún programación dinámica



4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

2. Formulación recursiva.

Ineficiencia:

- ✓Un problema de tamaño n se reduce a dos sub-problemas de tamaño (n-1).
- ✓Cada uno de los dos sub-problemas se reduce a otros dos…
- ✓Por tanto, se obtiene un algoritmo exponencial.

Sin embargo, el número total de sub-problemas a resolver no es tan grande:

- ➢La función recursiva B tiene dos parámetros: el primero puede tomar n valores distintos y el segundo, P valores.
- ➢Luego sólo hay n*P problemas diferentes.

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

3. Buscar estructura de datos reutilización resultados recurrencias.

   - La función recursiva B tiene dos parámetros: el primero puede tomar n valores distintos y el segundo, P valores.
   - Luego sólo hay n*P problemas diferentes.

Puesto que la solución recursiva está generando y resolviendo el mismo problema muchas veces, para evitar la repetición de cálculos, las soluciones de los sub-problemas se deben almacenar en una tabla.

Se puede construir una tabla B de tamaño (n+1)*(P+1), donde la fila 0 y la columna 0 son los casos base, con valor 0.
La tabla se rellena calculando la función B(k, p), recorriendo k de 1 a n, en nuestro caso de forma ascendente.

4 – Ejemplos: El problema de la Mochila 0/1

Planteamiento programación dinámica:

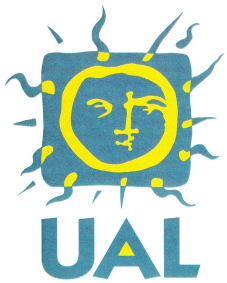
Ejemplo de tabla B:

Mochila de tamaño P = 11
Número de objetos n=5 (ordenados por peso)
Solución óptima {3,4} B = 40

Objeto	Valor	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

B

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40



4 – Ejemplos: El problema de la Mochila 0/1

Esquema algorítmico programación dinámica:

```
Método knapsack (entero [1..n] p,b, entero P):int[ ][ ]
    int [n][P] B
    para w=0 hasta P          /* Inicializamos la tabla sin objetos
            B[0][w]=0
    fpara
    para k=1 hasta n          /* Analizamos el efecto de contar con cada elemento
            para w=0 hasta p[k]-1              /* Mochilas menores tamaño elem.
                    B[k][w] = B[k-1][w];
            fpara
            para w=p[k] hasta P                /* Mochilas cabe elemento
                    B[k][w] = max ( B[k-1][w−p[k]]+b[k], B[k-1][w] );
            fpara
    fpara
 retorna B
fmétodo
```

4 – Ejemplos: El problema de la Mochila 0/1

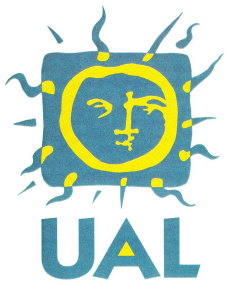
Recuperación de la solución óptima a partir de B[k,p]:

Calculamos la solución para B[k][p] utilizando el siguiente algoritmo:

- Si B[k][p] = B[k-1][p], entonces el objeto k no se selecciona y se seleccionan los objetos correspondientes a la solución óptima para k-1 objetos y una mochila de capacidad p (w en el algoritmo): la solución para B[k-1][p].

- Si B[k][p] ≠ B[k-1][p], se selecciona el objeto k, y además los objetos correspondientes a la solución óptima para k-1 objetos y una mochila de capacidad p-p[k]: la solución para B[k-1][p-p[k]].

Partimos del valor de la tabla B más alto B[n][P], beneficio máximo.

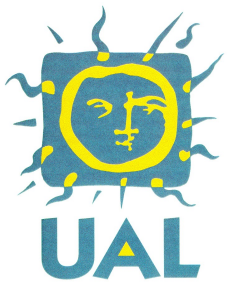


4 – Ejemplos: El problema de la Mochila 0/1

Resultados: Recuperación de la composición de la mochila

```
método test(entero j,c): conjunto enteros
    conjunto enteros sol
    sol = Ø
    si j>0 entonces
            si c<p[j] entonces
                  test(j-1,c)
             sino
                  si (B[j-1,c-p[j]]+b[j]>B[j-1,c]) entonces
                        test(j-1,c-p[j]);
                        sol = sol U {j}
                  sino
                        test(j-1,c)
                  fsi
            fsi
    fsi
fmétodo
```

```
Método objetos(entero[1..n] b,p; entero P;
                        entero [0..n][0..P] B): conjunto enteros
      sol = test(n, P)
fmétodo
```



4 – Ejemplos: El problema de la Mochila 0/1

Eficiencia de la solución con programación dinámica:

- Cada componente de la tabla B se calcula en tiempo constante, luego el coste de construcción de la tabla es O(n*P).

- El algoritmo test se ejecuta una vez por cada valor de j, desde n descendiendo hasta 0, luego su coste es O(n).

- Si P es muy grande, entonces esta solución no es buena.

- Tiempo de ejecución: Θ(n*P) "Pseudo-polinómico" (no es polinómico sobre el tamaño de la entrada, es decir, sobre el número de objetos).

- Recordemos que el problema de la mochila 0/1 es NP.

4 – Ejemplos: El problema del Viajante

Planteamiento formal:

Dado un grafo dirigido, con arcos (aristas) de longitud no negativa, se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes (circuito *hamiltoniano)*.

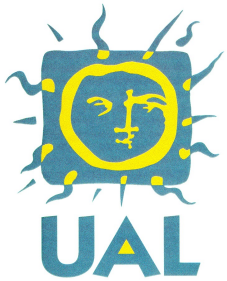
Sea:

$G = \langle V, A \rangle$ un grafo dirigido.

$V = \{1, 2, \ldots, n\}$ el conjunto de sus vértices.

L_{ij}, longitud de la arista $(i, j) \in A$. $L_{ij} = \infty$ si no existe arista (i, j).

Se trata de encontrar un recorrido de longitud mínima para un viajante que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.



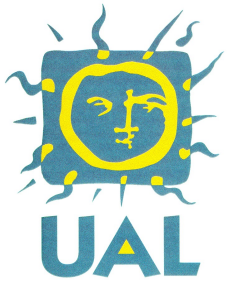
4 – Ejemplos: El problema del Viajante

Planteamiento formal: Principio de optimalidad

Solución del problema: sucesión de decisiones que verifica el principio de optimalidad. Idea: construir una solución mediante la búsqueda sucesiva de recorridos mínimos de tamaño 1, 2, 3, etc.

Supongamos que el circuito buscado empieza en el vértice 1. Se compondrá de la arista (1, j), con $j \neq 1$, seguida de un camino de j a 1, que pase una vez por cada vértice del subconjunto $V - \{1, j\}$.

Principio de optimalidad: si el circuito es óptimo, el camino de j a 1 debe serlo también, pues si no lo fuese llegaríamos a una contradicción. Si no lo fuese y existiera otro camino mejor, incluyendo a éste en el recorrido original, obtendríamos un circuito mejor que el óptimo, lo cual es imposible. Luego se cumple el principio de optimalidad.



4 – Ejemplos: El problema del Viajante

Planteamiento formal: Relación de recurrencia

Sea $S \subseteq V - \{1\}$ un subconjunto de vértices, e $i \in V - S$ un vértice; llamamos g(i, S) a la longitud del camino mínimo desde i hasta 1, que pase exactamente una vez por cada vértice de S.

Entonces, la longitud del circuito óptimo será:

$$g(1, V - \{1\}) = \min \{ L_{1j} + g(j, V - \{1, j\}) \}, \text{ con } 2 \leq j \leq n.$$

Esto se puede generalizar:

$$g(i, V - S) = \min \{ L_{ij} + g(j, S)\backslash\{j\} \}, \text{ con } j \in S. \qquad (**)$$

Teniendo finalmente (casos base):

$$g(i, \emptyset) = L_{i1} \text{ , con } i = 2, 3, \ldots n$$

4 – Ejemplos: El problema del Viajante

Planteamiento formal: recurrencia, método de resolución

$$g(i, V - S) = \min \{ L_{ij} + g(j, S)\backslash\{j\} \}, \text{ con } j \in S. \qquad (**)$$

$$g(i, \emptyset) = L_{i1} \text{ , con } i = 2, 3, \ldots n$$

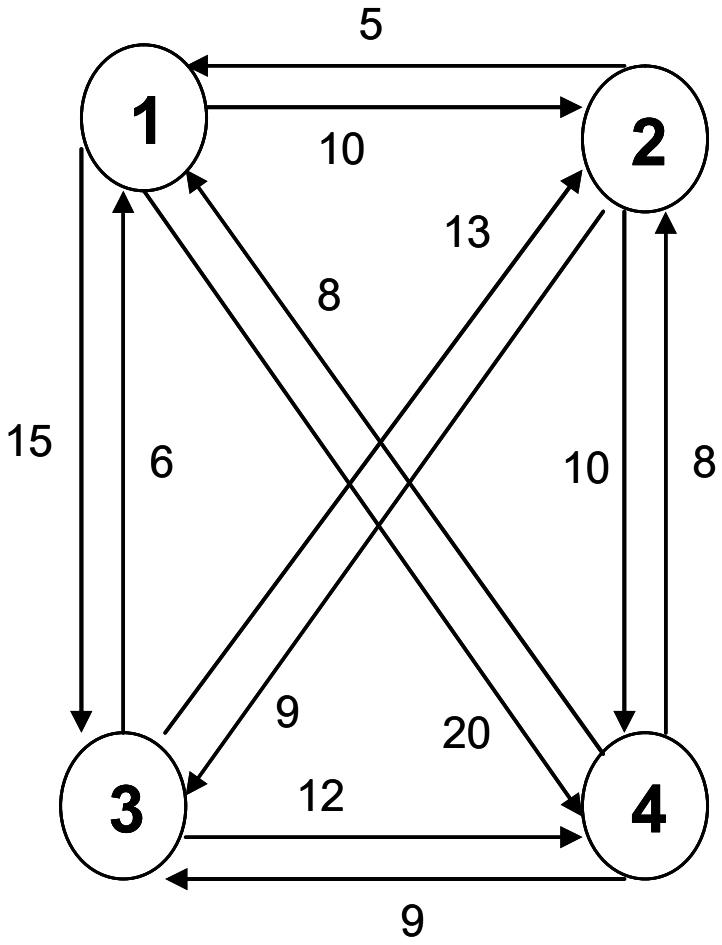
a) Usar (**) y calcular g para todos los conjuntos S con un solo vértice (distinto del 1).

b) Volver a usar (**) y calcular g para todos los conjuntos S de dos vértices (distintos del 1) y así sucesivamente.

c) Cuando se conoce el valor de g para todos los conjuntos S a los que sólo les falta un vértice (distinto del 1), basta calcular g(1, V - {1}).

4 – Ejemplos: El problema del Viajante

Planteamiento formal: ejemplo



Inicialización:

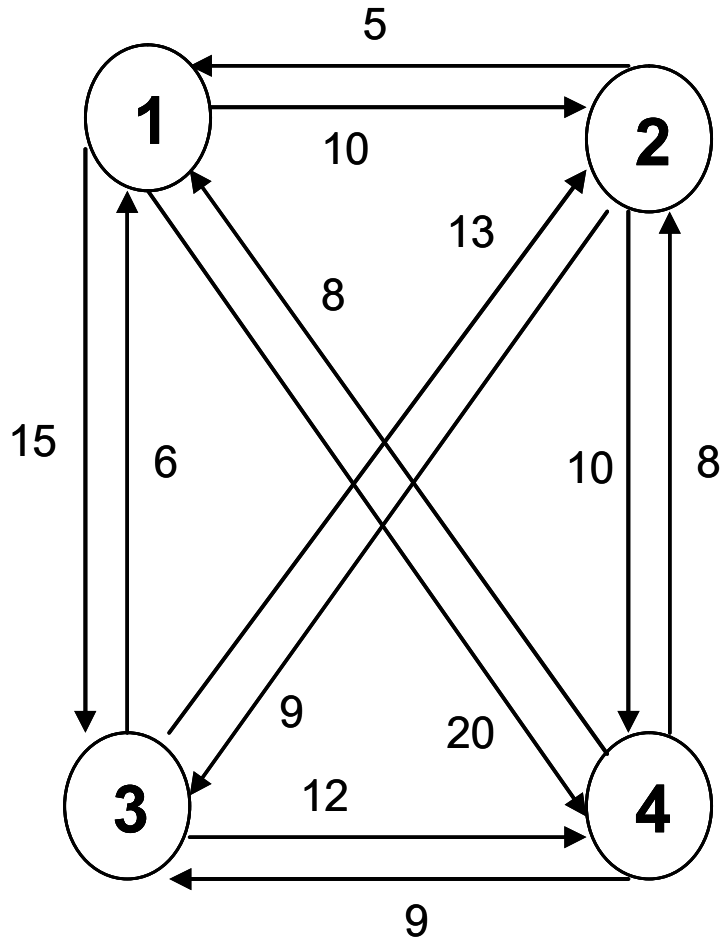
$$g(2, \emptyset) = L_{21} = 5$$

$$g(3, \emptyset) = L_{31} = 6$$

$$g(4, \emptyset) = L_{41} = 8$$

4 – Ejemplos: El problema del Viajante

Planteamiento formal: ejemplo



Conjuntos con un vértice es S:

$g(2, \{3\}) = L_{23} + g(3, \emptyset) = 9 + 6 = 15$

$g(2, \{4\}) = L_{24} + g(4, \emptyset) = 10 + 8 = 18$

$g(3, \{2\}) = L_{32} + g(2, \emptyset) = 13 + 5 = 18$

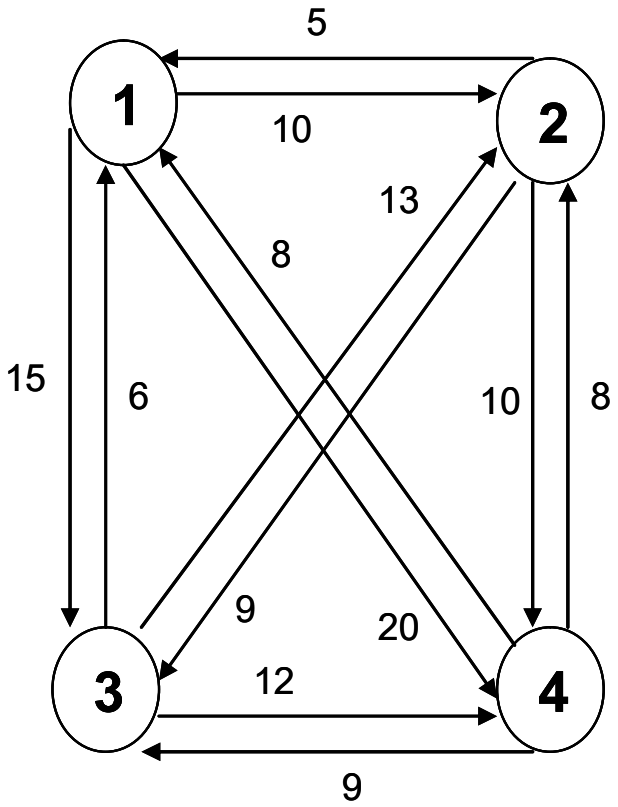
$g(3, \{4\}) = L_{34} + g(4, \emptyset) = 12 + 8 = 20$

$g(4, \{2\}) = L_{42} + g(2, \emptyset) = 8 + 5 = 13$

$g(4, \{3\}) = L_{43} + g(3, \emptyset) = 9 + 6 = 15$

4 – Ejemplos: El problema del Viajante

Planteamiento formal: ejemplo



Conjuntos con dos vértices es S:

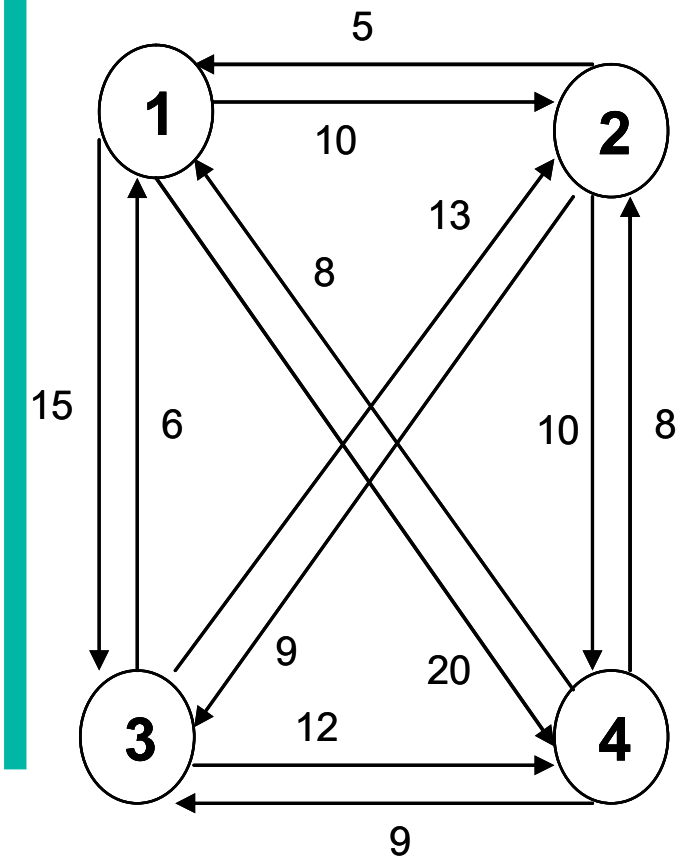
$g(2, \{3, 4\}) =$
$\min \{ L_{23} + g(3, \{4\}), L_{24} + g(4, \{3\}) \} =$
$\min \{ 9 + 20, 10 + 15 \} = \min \{ 29, 25 \} = 25$

$g(3, \{2, 4\}) =$
$\min \{ L_{32} + g(2, \{4\}), L_{34} + g(4, \{2\}) \} =$
$\min \{ 13 + 18, 12 + 13 \} = \min \{ 31, 25 \} = 25$

$g(4, \{2, 3\}) =$
$\min \{ L_{42} + g(2, \{3\}), L_{43} + g(3, \{2\}) \} =$
$\min \{ 8 + 15, 9 + 18 \} = \min \{ 23, 27 \} = 23$

4 – Ejemplos: El problema del Viajante

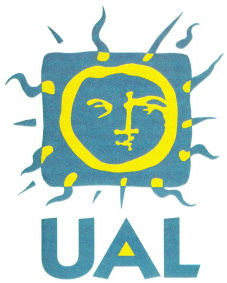
Planteamiento formal: ejemplo



Conjuntos con tres vértices es S, es decir la solución del problema:

$$g(1, \{2, 3, 4\}) = \min\{ L_{12} + g(2, \{3, 4\}),\ L_{13} + g(3, \{2, 4\}),\ L_{14} + g(4, \{2, 3\}) \} =$$
$$\min \{ 10 + 25, 15 + 25, 20 + 23 \} =$$
$$\min \{ 35, 40, 43 \} = 35$$

Luego la longitud del circuito óptimo, con origen y final en el vértice 1, es 35.



4 – Ejemplos: El problema del Viajante

Planteamiento formal: Recuperación del camino

Si además se quiere saber *cómo se construye el circuito óptimo*, hay que utilizar una función adicional J(i, S), que es el valor de j que minimiza g(i, S) al aplicar la fórmula (**).

De esta forma, se mantiene registro de las decisiones que se han ido tomando a lo largo del algoritmo.

4 – Ejemplos: El problema del Viajante

Planteamiento formal: Recuperación del camino, ejemplo

Circuitos con un vértice.

$J(2, \{3\}) = 3$

$J(2, \{4\}) = 4$

$J(3, \{2\}) = 2$

$J(3, \{4\}) = 4$

$J(4, \{2\}) = 2$

$J(4, \{3\}) = 3$

Circuitos con dos vértices.

$J(2,\{3,4\}) = 4$

$J(3,\{2,4\}) = 4$

$J(4,\{2,3\}) = 2$

Circuitos con tres vértices Circuito final.

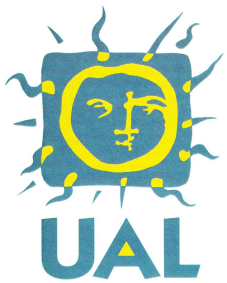
$J(1,\{2,3,4\}) = 2$

4 – Ejemplos: El problema del Viajante

Planteamiento formal: Recuperación del camino, ejemplo

$1 \rightarrow J(1,\{2,3,4\}) = 2$	Partiendo del origen, al valor de J para el cálculo final, donde i=origen y S=subconjunto con todos los vértices menos el origen.
$2 \rightarrow J(2,\{3,4\}) = 4$	Partiendo del vértice encontrado en el paso anterior, al valor de J para i=vértice del paso anterior y S=subconjunto donde ya no aparece dicho vértice.
$4 \rightarrow J(4,\{3\}) = 3$	Análogo al paso anterior.
$3 \rightarrow 1$	En el último paso, como el subconjunto S se queda vacío, se regresa al vértice origen.

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$



4 – Ejemplos: El problema del Viajante

Esquema algorítmico recursivo (función g):

```
{Se supone el acceso al grafo, global}
método g(nodo/entero i, conjunto nodos/enteros S): entero
    entero masCorto,distancia,j
    si S=Ø   entonces
        devuelve L[i,1]
    sino
        masCorto:=∞;
        para todo j en S hacer
           distancia:=L[i,j]+g(j,S\{j});
           si distancia < masCorto entonces
                  masCorto = distancia
           fsi
        fpara;
        devuelve masCorto
    fsi
finMetodo
```

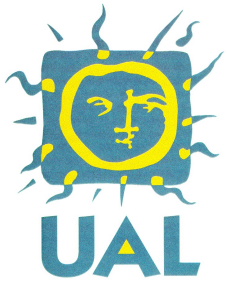
Se calcula repetidas veces el mismo valor de g: O((n-1)!).

4 – Ejemplos: El problema del Viajante

```
{se usa una tabla gtab cuyos elementos se inicializan con -1}
método g(i,S): entero
    entero masCorto,distancia,j
    si S=Ø  entonces  devuelve L[i,1]
    sino
        si gtab[i,S]≥0 entonces  devuelve gtab[i,S]
        sino
            masCorto:=∞;
            para todo j en S hacer
                distancia:=L[i,j]+g(j,S\{j});
                si distancia<másCorto entonces
                        masCorto:=distancia
                fsi
            fpara;
            gtab[i,S]:=masCorto;
            devuelve masCorto
        fsi
     fsi
fmétodo
```

Esquema de solución iterativo utilizando tablas:

Completar con la tabla J



4 – Ejemplos: El problema del Viajante

Esquema de solución iterativo utilizando tablas:

Obsérvese la diferencia que existe entre la estrategia de este algoritmo y la técnica voraz.

- En los algoritmos voraces, se ha de escoger una de las posibles opciones en cada paso, y una vez tomada –o descartada–, ya no vuelve a ser considerada nunca. Son algoritmos que no guardan "historia", y por tanto no siempre funcionan.

- En la programación dinámica, la solución al problema total se va construyendo de otra forma: a partir de las soluciones óptimas para problemas más pequeños.

4 – Ejemplos: El problema del Viajante

Esquema de solución iterativo utilizando tablas:

El diseño con programación dinámico aquí realizado tiene un serio inconveniente:

- Su implementación utilizando una estructura de datos que permita reutilizar los cálculos. Tal estructura debería contener las soluciones intermedias necesarias para el cómputo de g(1, V - {1}).

- La tabla debe tener n filas, y 2^n columnas, pues éste es el cardinal de las partes del conjunto V, que son todas las posibilidades que puede tomar el segundo parámetro de g en su definición.

4 – Ejemplos: El problema del Viajante

Esquema de solución iterativo utilizando tablas: eficiencia

- Calculo de g(j,Ø): n-1 consultas a la tabla.

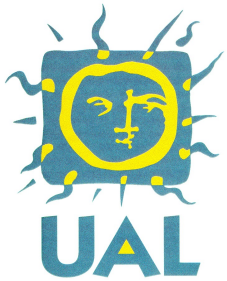
- Calculo de los g(j,S), tales que 1 < = card(S)=k < = n-2, combinaciones de n-2 elementos tomados de k en k, con n-1 alternativas posibles:

$$(n-1)\begin{bmatrix} n-2 \\ k \end{bmatrix} k \quad \text{sumas en total}$$

- Calculo de g(1,V\{1}): n-1 sumas.

Eficiencia total:

$$O(\ 2(n-1) + \sum_{k=1}^{n-1} (n-1)\, k \begin{bmatrix} n-2 \\ k \end{bmatrix}\ ) = O(n^2 2^n)$$



4 – Ejemplos: Planificación de tareas.

Planteamiento formal: Selección de actividades con pesos

Dado un conjunto C de n tareas o actividades, con:

s_i = tiempo de comienzo de la actividad i

f_i = tiempo de finalización de la actividad i

v_i = valor (o peso) de la actividad i

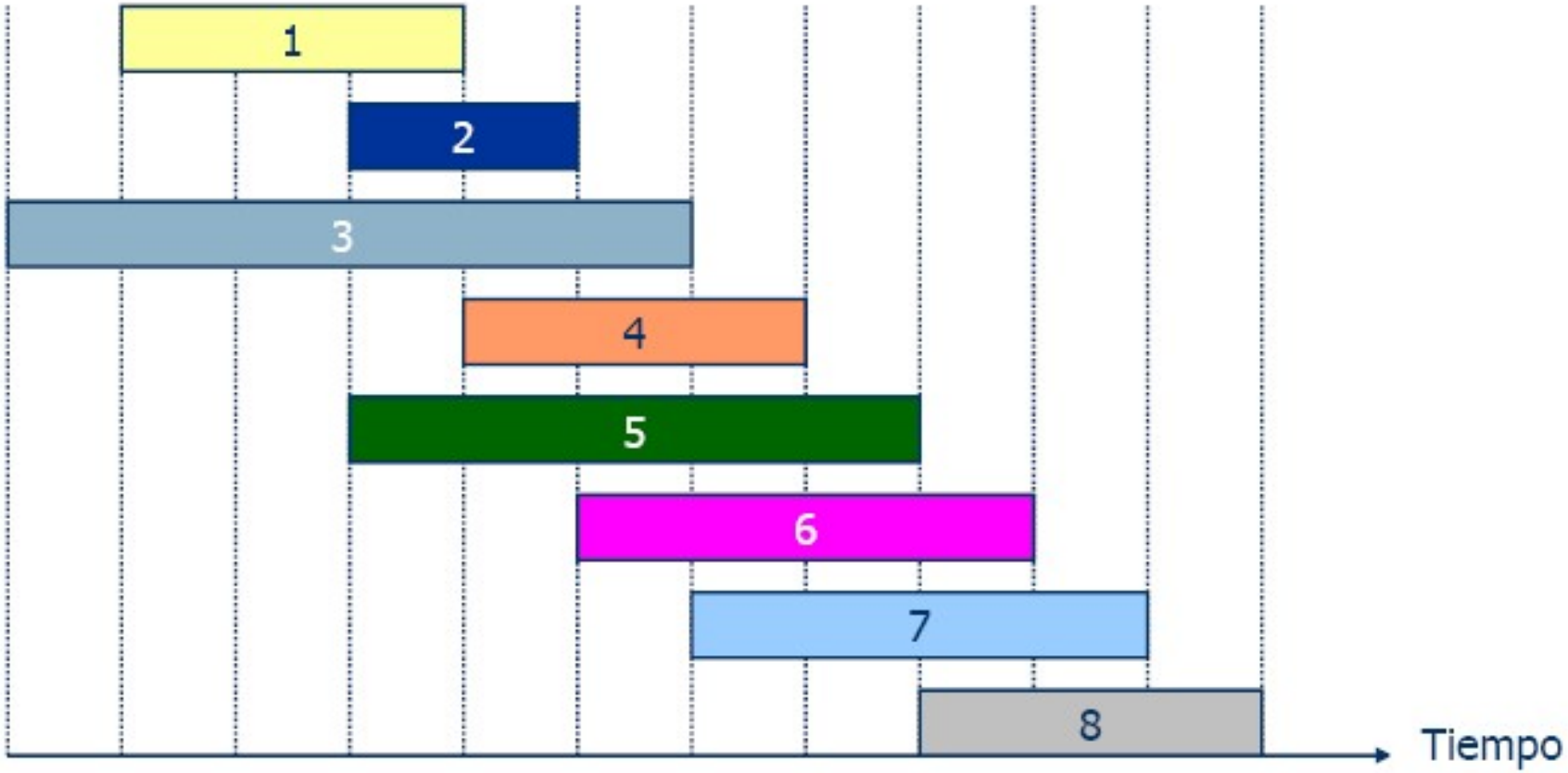
Se trata de encontrar el subconjunto S de actividades compatibles de peso máximo (esto es, un conjunto de actividades que no se solapen en el tiempo y que, además, nos proporcione un valor máximo).

Algoritmo "greedy" no funciona de forma general

4 – Ejemplos: Planificación de tareas.

Selección de actividades con pesos, planteamiento solución

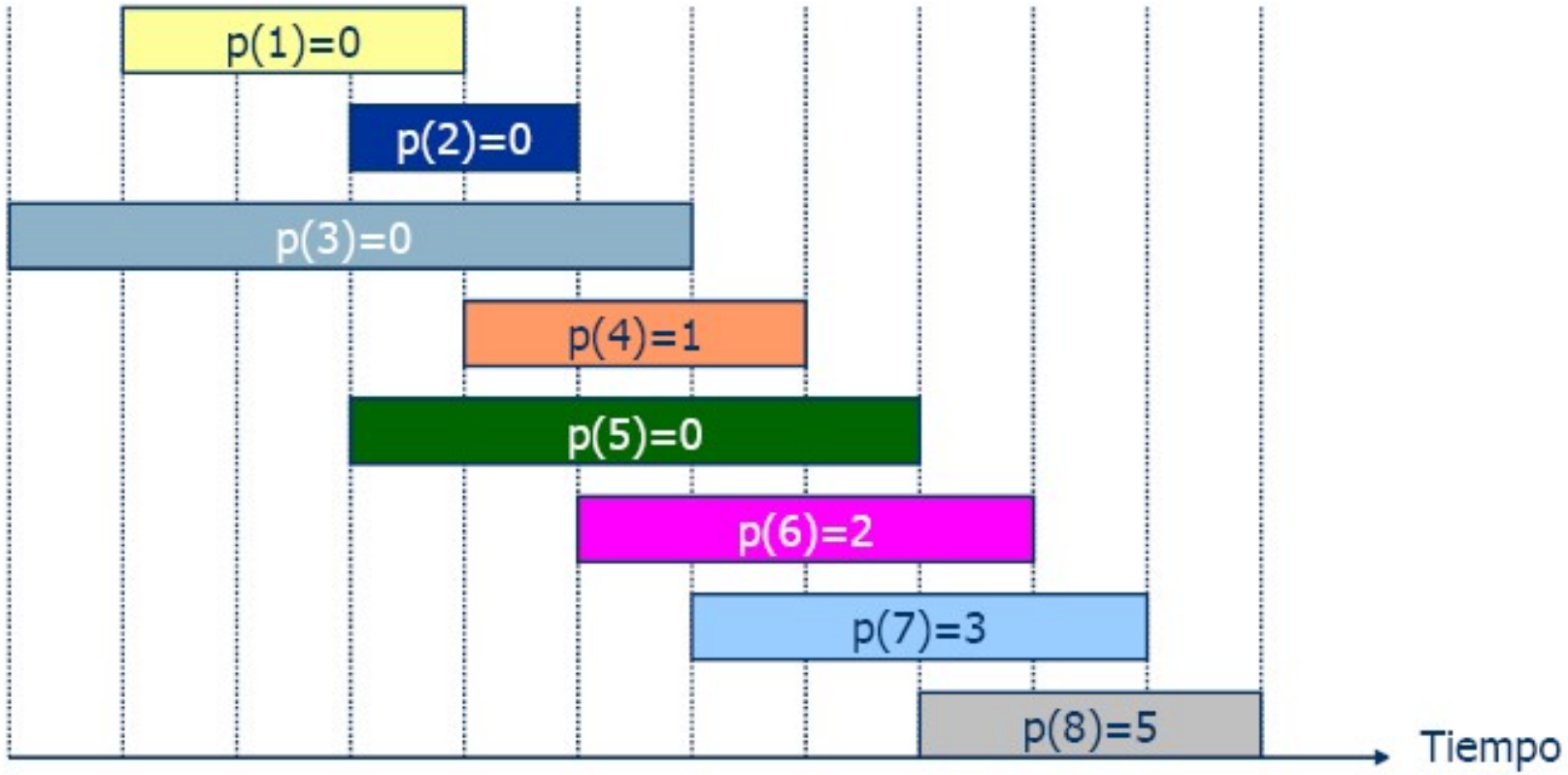
Ordenamos las actividades por hora de terminación

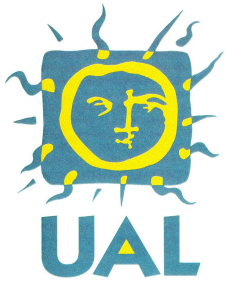


4 – Ejemplos: Planificación de tareas.

Selección de actividades con pesos, planteamiento solución

Podemos definir p(j) como el mayor índice i < j tal que la actividad i es compatible con la actividad j:





4 – Ejemplos: Planificación de tareas.

Definición recursiva de la solución:

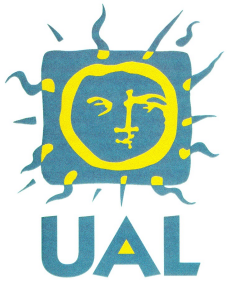
$$OPT(j) = \begin{cases} 0 & si \quad j = 0 \\ \max\{v(j) + OPT(p(j)), OPT(j-1)\} & si \quad j > 0 \end{cases}$$

Caso 1: el máximo es v(j) + OPT (p(j)), es decir, se elige la actividad j.

- La actividad j mejora el valor acumulado que se tenía.
- No se pueden escoger actividades incompatibles > p(j).
- La solución incluirá la solución óptima para p(j).

Caso 2: el máximo es OPT(j-1), es decir, no se elige la actividad j.

- La actividad j no mejora el valor acumulado que se tenía.
- La solución coincidirá con la solución óptima para las primeras (j-1) actividades.



4 – Ejemplos: Planificación de tareas.

Implementación iterativa:

```
método selecActivConPesos (vector [1..n] actividades A): tipoSolución
    Ordenar A[1] .. A[n] por tiempo creciente de finalización   // O(n log n)
    Calcular p[1] .. p[n]                                      // O(n log n)
    OPT[0] = 0                                                 // Caso base O(1)
    para i desde 1 hasta n hacer                               // O(n)
        OPT[i] = max (valor[i]+OPT[ p[i] ], OPT[i-1] )
    fpara
    retorna Solución(OPT)        // Reconstruir solución a partir de OPT O(n)
fmétodo
```

4 – Ejemplos: Planificación de tareas.

Implementación iterativa: ejemplo anterior

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
valor	5	7	10	4	9	8	3	2

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
p	0	0	0	1	0	2	3	5

Resultado de aplicar el algoritmo:

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
OPT	0	5	7	10	10	10	15	15	15

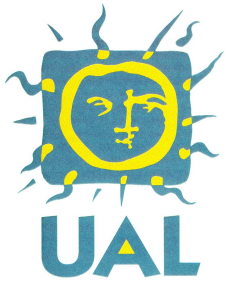
4 – Ejemplos: Planificación de tareas.

Implementación iterativa: ejemplo anterior

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
OPT	0	5	7	10	10	10	15	15	15

Reconstrucción de la solución: secuencia de decisiones

tarea 1: el máximo es valor[i] + OPT [p[i]] → Elegir
tarea 2: el máximo es valor[i] + OPT [p[i]] → Elegir
tarea 3: el máximo es valor[i] + OPT [p[i]] → Elegir
tarea 4: el máximo es OPT [i-1] → No elegir
tarea 5: el máximo es OPT [i-1] → No elegir
tarea 6: el máximo es valor[i] + OPT [p[i]] → Elegir
tarea 7: el máximo es OPT [i-1] → No elegir
tarea 8: el máximo es OPT [i-1] → No elegir



4 – Ejemplos: Planificación de tareas.

Implementación iterativa: ejemplo anterior

tarea 1: el máximo es valor[i] + OPT [p[i]]	→	Elegir
tarea 2: el máximo es valor[i] + OPT [p[i]]	→	Elegir
tarea 3: el máximo es valor[i] + OPT [p[i]]	→	Elegir
tarea 4: el máximo es OPT [i-1]	→	No elegir
tarea 5: el máximo es OPT [i-1]	→	No elegir
tarea 6: el máximo es valor[i] + OPT [p[i]]	→	Elegir
tarea 7: el máximo es OPT [i-1]	→	No elegir
tarea 8: el máximo es OPT [i-1]	→	No elegir

Reconstrucción de la solución: marcha atrás…

Último elegir: 6

p(6) = 2 --- Estaba elegir, se elige 2
(Si no elegir, se iría atrás, 1….)
p(2) = 0 --- Ya no se toman más.

(2,6)

4 – Ejemplos: Planificación de tareas.

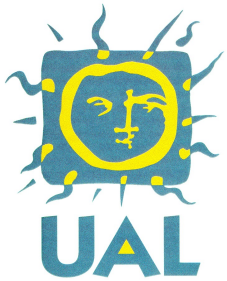
Implementación iterativa: otro ejemplo

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
valor	5	7	10	4	9	8	3	12

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
p	0	0	0	1	0	2	3	5

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
OPT	0	5	7	10	10	10	15	15	22

Rehacer el mecanismo (3,8)



4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos

Árbol binario de búsqueda: La clave de todo nodo es mayor que las de sus descendientes izquierdos y menor que las de sus descendientes derechos.

Problema: Se tiene un conjunto de palabras (claves) distintas, ordenadas alfabéticamente:

$$w_1 < w_2 < \ldots < w_n$$

que deben almacenarse en un árbol binario de búsqueda.

Se conoce la probabilidad p_i, $1 \leq i \leq n$, con la que se pide buscar la palabra w_i y su información asociada.

Se quiere construir un árbol binario de búsqueda para guardar las palabras, que minimice el coste, es decir, el número medio de comparaciones para encontrar una palabra.

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos

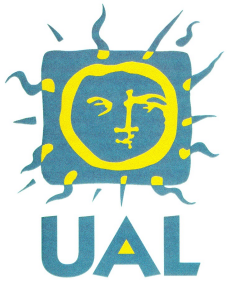
La profundidad de la raíz es 0, la de sus hijos es 1, etc.

Si construimos un árbol en el que la palabra w_i está en un nodo de profundidad d_i, donde $1 \leq i \leq n$, entonces se necesitan $d_i + 1$ comparaciones para encontrarla.

Por tanto, el número medio de comparaciones para encontrar una palabra es:

$$C = \sum_{i=1}^{n} p_i \, (d_i + 1)$$

Ésa es, pues, la función que queremos minimizar. El árbol binario de búsqueda óptimo para nuestra secuencia de palabras, será aquél donde el coste C sea mínimo.



4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

Solución del problema mediante *programación dinámica*:

Supongamos que queremos colocar en un árbol binario de búsqueda la secuencia de palabras (ordenadas):

$$w_{izq}, w_{izq-1}, \ldots, w_{der-1}, w_{der}$$

Supongamos que el árbol binario de búsqueda óptimo tiene a w_i como raíz, donde se cumple que izq < i < der. Entonces, el subárbol izquierdo debe contener las siguientes palabras: w_{izq} .. w_{i-1}, y el subárbol derecho, w_{i+1} .. w_{der}, por la propiedad de árbol binario de búsqueda.

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

Si se cumple el principio de *optimalidad*, es decir, que todos los subárboles de un árbol óptimo son óptimos con respecto a las claves que contienen, entonces, los subárboles izquierdo y derecho anteriores deben ser también óptimos, pues, de otra forma, podrían ser sustituidos por subárboles óptimos, lo cual daría una mejor solución para la secuencia completa $w_{izq}, \ldots, w_{der}$.

Así, podemos encontrar una fórmula que exprese el coste del árbol binario de búsqueda óptimo ($C_{izq\ der}$), en función del coste de su subárbol izquierdo ($C_{izdo\ i-1}$) y del coste de su subárbol derecho ($C_{i+1\ der}$).

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

Para los sub-árboles hijos, el *coste relativo a su propia raíz*, vale:

$$C_{izdo\ i-1} = \sum_{j=izdo}^{i-1} p_j \cdot n_j$$

$$C_{i+1\ der} = \sum_{j=i+1}^{der} p_j \cdot n_j$$

donde:

p_j = probabilidad de acceder al nodo j

n_j = nº de comparaciones hasta el nodo j

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

Si se añade una raíz w_i sobre ambos sub-árboles hijos, estamos añadiendo 1 al número de comparaciones para llegar hasta cada nodo de los sub-árboles hijos, de modo que el *coste desde la raíz* será:

$$C_{\text{izdo i-1 desde raiz}} = \sum_{j=izdo}^{i-1} p_j \cdot (n_j + 1) = \sum_{j=izdo}^{i-1} p_j \cdot n_j + \sum_{j=izdo}^{i-1} p_j$$

$$C_{\text{i+1 der desde raiz}} = \sum_{j=i+1}^{der} p_j \cdot (n_j + 1) = \sum_{j=i+1}^{der} p_j \cdot n_j + \sum_{j=i+1}^{der} p_j$$

Más el coste de la propia raíz que es: $C_i = p_i \cdot 1 = p_i$

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

- Coste del árbol completo:

$$C_{izq\ der} = C_{izdo\ i-1\ desde\ raíz} + C_i + C_{i+1\ der\ desde\ raíz}$$

$$= \sum_{j=izdo}^{i-1} p_j \cdot n_j + \sum_{j=izdo}^{i-1} p_j + p_i + \sum_{j=i+1}^{der} p_j \cdot n_j + \sum_{j=i+1}^{der} p_j$$

$$= \sum_{j=izdo}^{i-1} p_j \cdot n_j + \sum_{j=i+1}^{der} p_j \cdot n_j + \sum_{j=izdo}^{i-1} p_j + p_i + \sum_{j=i+1}^{der} p_j$$

- Coste árbol completo en función de los árboles izquierdo y derecho:

$$C_{izq\ der} = C_{izdo\ i-1} + C_{i+1\ der} + \sum_{j=izdo}^{der} p_j$$

Está es la expresión recursiva que permite construir la solución.

4 – Ejemplos: Otros ejemplos.

Árboles binarios de búsqueda óptimos: planteamiento solución

- Construcción de la tabla:

  + Primera fila, óptimos con árboles con sólo una palabra (n).

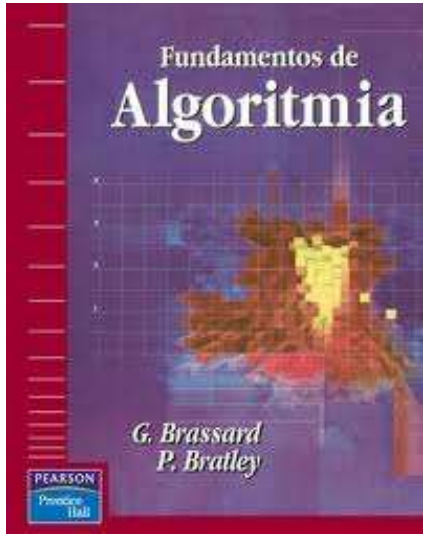
  + Segunda fila, óptimos con árboles con dos palabras (n-1).

  + Tercera fila, óptimos con árboles de tres palabras (n-2).

  + ……

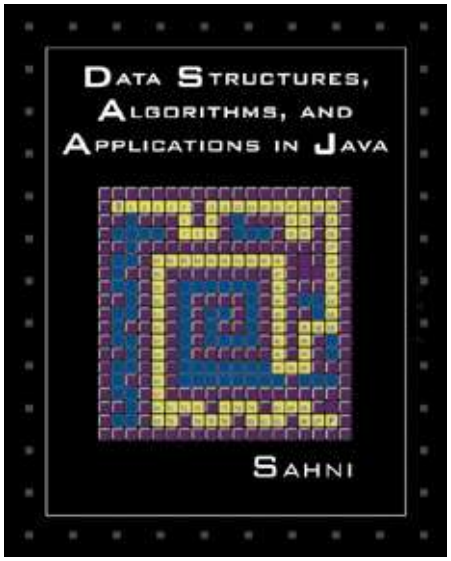
  Se almacenan los óptimos para secuencias de palabras de logintudes1, 2, 3,….n, cuando llegamos al n, tenemos el árbol óptimo.

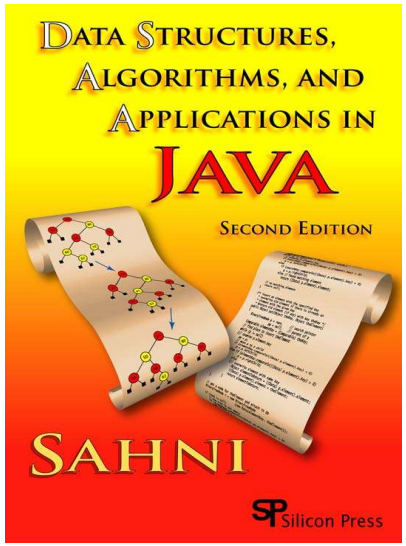
Material a estudiar



Fundamentos de algoritmia
G. Brassard, T. Bratley
Prentice Hall, D.L. 2004

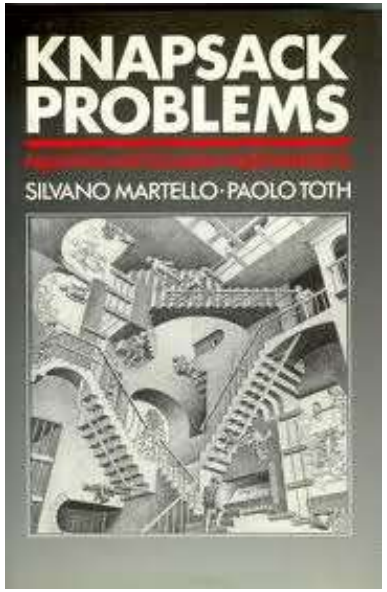
Biblioteca: 519 BRA fun
Capítulo 8.





Data Structures, Algorithms, and Applications in Java
Sartaj Sahni
McGraw Hill, 2000 / Silicom Press, 2005
Biblioteca: Agotado – Esperando 2ª edición. Capítulo 20.

Material a estudiar



KNAPSACK PROBLEMS
S. Martello, P. Toth
John Wiley & Sons, 1990
Descatalogado.
Libre acceso en:
http://www.or.deis.unibo.it/knapsack.html
(comprobado con fecha 10/03/2016)

Ejercicios recomendados - Libres

0. Implementar el esquema algorítmico de Floyd y las opciones alternativas (fuerza bruta).

Se ha de:

- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba generados aleatoriamente (dado el tamaño del problema).
- Almacenar juegos de prueba, resultados y tiempos de ejecución.
- Comparar los resultados de fuerza bruta, de Dijkstra sobre todos los nodos (ejercicio 0 del tema anterior) y Floyd.
- Analizar la eficiencia obtenida empíricamente frente a la teórica.

Ejercicios recomendados - Libres

1. Comparar con distintos ejemplos la solución del problema de la mochila 0/1 utilizando el esquema voraz y la programación dinámica.

   Ha de:

   - Presentar los esquemas algorítmicos detallados (pseudocódigo).
   - Seleccionar las estructuras de datos adecuadas para almacenar los datos.
   - Implementar el correspondiente código en Java.
   - Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba generados aleatoriamente (dado el tamaño del problema).
   - Almacenar juegos de prueba, resultados y tiempos de ejecución.
   - Analizar la eficiencia obtenida empíricamente frente a la teórica.

Ejercicios recomendados - Libres

2. Comparar las soluciones voraces (con distintas funciones de selección) y la programación dinámica para el problema de selección de actividades con pesos.
   - Presentar los esquemas algorítmicos detallados (pseudocódigo).
   - Seleccionar las estructuras de datos adecuadas para almacenar los datos (cuidado con el esquema de programación dinámica.)
   - Implementar el correspondiente código en Java.
   - Almacenar juegos de prueba, resultados y tiempos de ejecución.
   - Analizar la eficiencia obtenida empíricamente frente a la teórica, realizando una comparación de eficiencia y resultado obtenido.

Ejercicios recomendados - Libres

3. Desarrollar con más detalle el problema de árboles binarios de búsqueda.

Se ha de:

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos (muy importante, mostrar el TAD y las estructuras a implementar.)
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba.
- Almacenar juegos de prueba, resultados y tiempos de ejecución.
- Analizar la eficiencia obtenida empíricamente frente a la teórica.