DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Programación Dinámica

Análisis y Diseño de Algoritmos

Programación Dinámica

- Introducción
- Elementos de la programación dinámica
  - Principio de optimalidad de Bellman
  - Definición recursiva de la solución óptima
  - Cálculo de la solución óptima
- Ejemplos
  - Multiplicación encadenada de matrices
  - Subsecuencia de mayor longitud (LCS)
  - Selección de actividades con pesos
  - Problema de la mochila
  - Caminos mínimos: Algoritmos de Floyd y Bellman-Ford
  - Distancia de edición
- Aplicaciones

Programación Dinámica

Esta técnica se aplica sobre problemas que presentan las siguientes características:

- **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 se resuelve más de una vez.

Programación Dinámica

Enfoque ascendente (bottom-up):

- Primero se calculan las soluciones óptimas para problemas de tamaño pequeño.
- Luego, utilizando dichas soluciones, encuentra soluciones a problemas de mayor tamaño.

Clave: Memorización

Almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente. De esa forma, se consigue un algoritmo más eficiente que la fuerza bruta, que resuelve el mismo subproblema una y otra vez.

Programación Dinámica

Memorización

Para evitar calcular lo mismo varias veces:

- Cuando se calcula una solución, ésta se almacena.
- Antes de realizar una llamada recursiva para un subproblema 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 las soluciones a los subproblemas son calculados (espacio vs. tiempo).

Programación Dinámica

Uso de la programación dinámica:

1. Caracterizar la estructura de una solución óptima.
2. Definir de forma recursiva la solución óptima.
3. Calcular la solución óptima de forma ascendente.
4. Construir la solución óptima a partir de los datos almacenados al obtener soluciones parciales.

Estrategias de diseño

Algoritmos greedy:

Se construye la solución incrementalmente, utilizando un criterio de optimización local.

Programación dinámica:

Se descompone el problema en subproblemas solapados y se va construyendo la solución con las soluciones de esos subproblemas.

Divide y vencerás:

Se descompone el problema en subproblemas independientes y se combinan las soluciones de esos subproblemas.

Principio de Optimalidad

Para poder emplear programación dinámica, una secuencia óptima debe cumplir la condición de que cada una de sus subsecuencias también sea óptima:

Dado un problema P con n elementos,
si la secuencia óptima es $e_1e_2...e_k...e_n$ entonces:

- $e_1e_2...e_k$ 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.

Principio de Optimalidad

En otras palabras:

La solución óptima de cualquier instancia no trivial de un problema es una combinación de las soluciones óptimas de sus subproblemas.

- Se busca la solución óptima a un problema como un proceso de decisión "multietápico".
- Se toma una decisión en cada paso, pero ésta depende de las soluciones a los subproblemas que lo componen.

Principio de Optimalidad

Un poco de historia: Bellman, años 50…

Enfoque 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.

Etimología: Programación dinámica = Planificación temporal

En una época "hostil" a la investigación matemática, Bellman buscó un nombre llamativo que evitase posibles confrontaciones:

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"

Richard E. Bellman: "Eye of the Hurricane: An Autobiography"

Principio de Optimalidad

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 solucíón óptima con respecto al estado resultante de la primera decisión".

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).

Principio de Optimalidad

Principio de Optimalidad de Bellman

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

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

¡Ojo!

El principio de optimalidad no nos dice que, si tenemos las soluciones óptimas de los subproblemas, entonces podamos combinarlas para obtener la solución óptima del problema original...

Principio de Optimalidad

Principio de Optimalidad de Bellman

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

Ejemplo: Cambio en monedas

- 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 (0.05 + 2) + (0.05 + 0.01) euros.

Sin embargo, sí que existe alguna forma de descomponer 0.13 euros de tal forma que las soluciones óptimas a los subproblemas nos den una solución óptima (p.ej. 0.11 y 0.02 euros).

Definición del problema...

Para aplicar programación dinámica:

1. Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que encontrar la "estructura" de la solución.

2. Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).

... y cálculo de la solución óptima

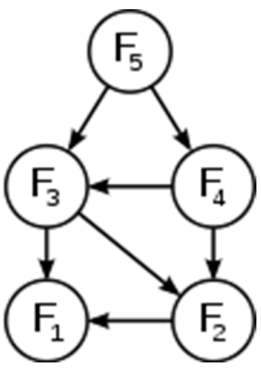
3. Se calcula el valor de la solución óptima utilizando un enfoque ascendente:
   - Se determina el conjunto de subproblemas que hay que resolver (el tamaño de la tabla).
   - Se identifican los subproblemas con una solución trivial (casos base).
   - Se van calculando los valores de soluciones más complejas a partir de los valores previamente calculados.

4. Se determina la solución óptima a partir de los datos almacenados en la tabla.

Programación Dinámica
Ejemplos

Sucesión de Fibonacci $fib(n) = fib(n-1) + fib(n-2)$

- Implementación recursiva: $O(\varphi^n)$
- Implementación usando programación dinámica: $\Theta(n)$




```
if (n == 0) return 0;
else if (n == 1) return 1;
else {
  previo = 0; actual = 1;
  for (i=1; i<n; i++) {
      fib = previo + actual;
      previo = actual; actual = fib;
  }
  return actual;
}
```

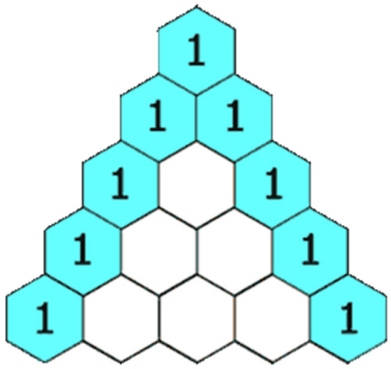
Programación Dinámica
Ejemplos

Números combinatorios:

Combinaciones de n sobre p

- Implementación inmediata… $\binom{n}{p} = \frac{n!}{p!(n-p)!}$
- Implementación usando programación dinámica… Triángulo de Pascal

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$



Programación Dinámica
Ejemplos

Números combinatorios:

Combinaciones de n sobre p

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

	p = 0	p = 1	p = 2	p = 3	p = 4	p = 5	p = 6	p = 7	p = 8	p = 9
n = 0	1									
n = 1	1	1								
n = 2	1	2	1							
n = 3	1	3	3	1						
n = 4	1	4	6	4	1					
n = 5	1	5	10	10	5	1				
n = 6	1	6	15	20	15	6	1			
n = 7	1	7	21	35	35	21	7	1		
n = 8	1	8	28	56	70	56	28	8	1	
n = 9	1	9	36	84	126	126	84	36	9	1

Orden de eficiencia: $\Theta(np)$

Programación Dinámica
Ejemplos

Números combinatorios:

Combinaciones de n sobre p

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

```
int combinaciones (int n, int p)
{
  for (i=0; i<=n; i++) {
      for (j=0; j<=min(i,p); j++) {
          if ((j==0) || (j==i))
             c[i][j] = 1
          else
             c[i][j] = c[i-1][j-1]+c[i-1][j];
      }
  }
  return c[n][p];
}
```

Orden de eficiencia: Θ(np) en tiempo, Θ(np) en espacio.

Programación Dinámica
Ejemplos

Números combinatorios:

Combinaciones de n sobre p

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

```
int combinaciones (int n, int p)
{
  b[0] = 1;
  for (i=1; i<=n; i++) {
      b[i] = 1;
      for (j=i-1; j>0; j--) {
          b[j] += b[j - 1];
      }
  }
  return b[p];
}
```

Orden de eficiencia: Θ(np) en tiempo, Θ(n) en espacio.

Programación Dinámica
Devolver el cambio...

Existen casos para los que no se puede aplicar el algoritmo greedy (por ejemplo, devolver 8 peniques con monedas de 6, 4 y 1 penique).

Definición recursiva de la solución

$$cambio(\{m_1,...,m_i\},C) = \min\begin{cases} cambio(\{m_1,...,m_{i-1}\},C) \\ 1+cambio(\{m_1,...,m_i\},C-m_i) \end{cases}$$

Cálculo de la solución con programación dinámica:

- Orden de eficiencia proporcional al tamaño de la tabla, O(Cn), donde n es el número de monedas distintas.

Programación Dinámica
Devolver el cambio...

$$cambio(\{m_1,...,m_i\},C) = \min\begin{cases} cambio(\{m_1,...,m_{i-1}\},C) \\ 1+cambio(\{m_1,...,m_i\},C-m_i) \end{cases}$$

	0	1	2	3	4	5	6	7	8
{1}	0	1	2	3	4	5	6	7	8
{1,4}	0	1	2	3	1	2	3	4	2
{1,4,6}	0	1	2	3	1	2	1	2	2

m={1,4,6}, C=8

Solución: 2 monedas, S={4,4}.

Programación Dinámica
Multiplicación encadenada de matrices

Propiedad asociativa del producto de matrices:

Dadas las matrices A_1 (10x100), A_2 (100x5) y A_3 (5x50),

- $(A_1A_2)A_3$ implica 7500 multiplicaciones
- $A_1(A_2A_3)$ implica 75000 multiplicaciones

"Parentizaciones" posibles:

$$P(n) = \begin{cases} 1 & si \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & si \quad n > 1 \end{cases} \qquad \Omega\left(\frac{4^n}{n^2}\right)$$

Programación Dinámica
Multiplicación encadenada de matrices

Si cada matriz A_k tiene un tamaño $p_{k-1}p_k$,
el número de multiplicaciones necesario será:

$$m(1,n) = m(1,k) + m(k+1,n) + p_0 p_k p_n$$

$A_1 \times A_2 \times A_3 \times \ldots \times A_k$ x $A_{k+1} \times A_{k+2} \times \ldots \times A_n$

De forma general:

$$m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} p_k p_j$$

Programación Dinámica
Multiplicación encadenada de matrices

Matriz A_k de tamaño $p_{k-1}p_k$

Definición recursiva de la solución óptima:

- Si i=j, entonces

$$m(i,j) = 0$$

- Si i≠j, entonces

$$m(i,j) = \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \}$$

$A_1 \times A_2 \times A_3 \times \ldots \times A_k$ x $A_{k+1} \times A_{k+2} \times \ldots \times A_n$

Programación Dinámica
Multiplicación encadenada de matrices

Implementación:

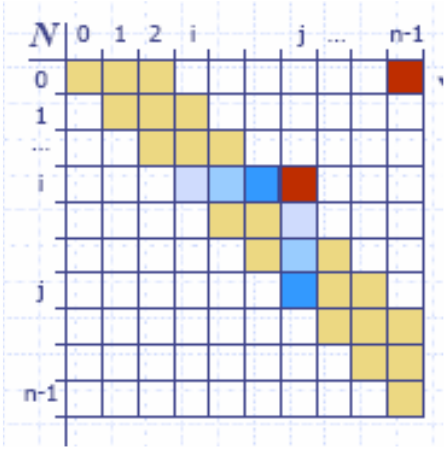
- Tenemos n^2 subproblemas distintos m(i,j).
- Para calcular m(i,j) necesitamos los valores almacenados en la fila i y en la columna j

$$m(i,j) = \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \}$$

- Para calcular cada valor necesitamos O(n), por lo que el algoritmo resultante es de orden $O(n^3)$.

Programación Dinámica
Multiplicación encadenada de matrices

Implementación:




```
for (i=1; i<=n; i++)
    m[i,i] = 0;

for (s=2; s<=n; l++)
    for (i=1; i<=n-s+1; i++) {
        j = i+s-1;
        m[i,j] = ∞;
        for (k=i; k<=j-1; k++) {
            q = m[i,k] + m[k+1,j] + p[i-1]*p[k]p[j]
            if (q < m[i,j]) {
               m[i,j] = q;
               s[i,j] = k;
            }
        }
    }
```


Programación Dinámica
Multiplicación encadenada de matrices

Implementación:

```
// Suponiendo que hemos calculado previamente s[i,j]…

MultiplicaCadenaMatrices (A, i, j)
{
  if (j>i) {
     x = MultplicaCadenaMatrices (A, i, s[i,j]);
     y = MultplicaCadenaMatrices (A, s[i,j]+1, j);
     return MultiplicaMatrices(x, y);
  } else {
     return A[i];
  }
}
```

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Problema:

Comparar dos cadenas y encontrar la subsecuencia común de mayor longitud.

Ejemplo:

```
Cadenas:
X = (A B C B D A B),
Y = (B D C A B A),

Subsecuencia común de mayor longitud:
LCS = (B C B A)

    X = ( A B   C   B D A B )
    Y = (   B D C A B   A   )
```


Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Un algoritmo de fuerza bruta compararía cualquier subsecuencia de X con los símbolos de Y:

Si |X|=m, |Y|=n, entonces hay 2^m subsecuencias de X que debemos comparar con Y (n comparaciones), por lo que nuestro algoritmo sería de orden **$O(n2^m)$**.

Sin embargo, LCS tiene subestructuras optimales:

Si buscamos la LCS para pares de prefijos de X e Y, podemos calcular la solución de nuestro problema…

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Si buscamos la LCS para pares de prefijos de X e Y, podemos calcular la solución de nuestro problema…

X_i	Prefijo de X de longitud i.
Y_j	Prefijo de Y de longitud j.
c(i,j)	Longitud de la LCS para X_i e Y_j.

$$c(i,j) = \begin{cases} c(i-1,j-1)+1 & si\ x[i]=y[j] \\ \max\{c(i-1,j), c(i,j-1)\} & en\ otro\ caso \end{cases}$$

La longitud de la subsecuencia de mayor longitud (LCS) de X e Y será **c(m,n),** con |X|=m, |Y|=n.

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Definición recursiva de la solución:

- Caso base:
  c(0,0) = 0 (Subcadena vacía)
  c(0,j) = c(i,0) = 0 (LCS de la cadena vacía y cualquier otra cadena)
- Cálculo recursivo:
  - Primer caso (x[i]=y[j]=s): Se emparejan ambos símbolos y la LCS aumenta de longitud: $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1})+\{s\}$.
  - Segundo caso (x[i]≠y[j]): No se pueden emparejar los símbolos x[i] e y[j], por lo que $LCS(X_i,Y_j)$ será la mejor entre $LCS(X_{i-1},Y_j)$ y $LCS(X_i,Y_{j-1})$.

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Implementación iterativa del algoritmo:

```
int[][] LCS (X, Y)
{
  for (i=0; i<=X.length; i++) c[i][0]=0;        // Y0
  for (j=1; j<=T.length; j++) c[0][j]=0;        // X0

  for (i=1; i<=X.length; i++)                   // Xi
      for (j=1; j<=Y.length; j++)               // Yj
          if ( X[i] == Y[j] )
             c[i][j] = c[i-1][j-1] + 1;
          else
             c[i][j] = max ( c[i-1][j], c[i][j-1] );

  return c;
}
```


Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Ejemplo:

X = (A B C B),
Y = (B D C A B),

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0					
B	2	0					
C	3	0					
B	4	0					

```
  for (i=0; i<=X.length; i++) c[i][0]=0;        // Y0
  for (j=1; j<=T.length; j++) c[0][j]=0;        // X0
```

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Ejemplo:

```
X = (A B C B),
Y = (B D C A B),
```

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	**0**	**0**	**0**		
B	2	0					
C	3	0					
B	4	0					

```
if ( X[i] == Y[j] )
   c[i][j] = c[i-1][j-1] + 1;
else
   c[i][j] = max ( c[i-1][j], c[i][j-1] );
```


Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Ejemplo:

```
X = (A B C B),
Y = (B D C A B),
```


		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	**0**	**0**	**0**	**1**	
B	2	0					
C	3	0					
B	4	0					


```
if ( X[i] == Y[j] )
   c[i][j] = c[i-1][j-1] + 1;
else
   c[i][j] = max ( c[i-1][j], c[i][j-1] );
```

Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Ejemplo:

```
X = (A B C B),
Y = (B D C A B),
```

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	**0**	**0**	**0**	**1**	**1**
B	2	0					
C	3	0					
B	4	0					

```
if ( X[i] == Y[j] )
   c[i][j] = c[i-1][j-1] + 1;
else
   c[i][j] = max ( c[i-1][j], c[i][j-1] );
```


Programación Dinámica
Subsecuencia de mayor longitud (LCS)

Ejemplo:

```
X = (A B C B),
Y = (B D C A B),

LCS = (B C B)

X = ( A B   C   B )
Y = (   B D C A B )
```

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	0	0	0	1	1
B	2	0	**1**	1	1	1	2
C	3	0	1	1	**2**	2	2
B	4	0	1	1	2	2	**3**

NOTA:

A partir del valor c[i][j] podemos determinar cómo se calculó éste y encontrar la LCS hacia atrás…

Programación Dinámica
Selección de actividades con pesos

Enunciado del problema

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

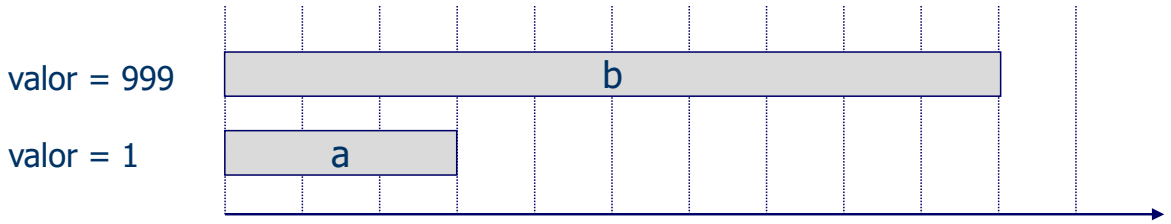
encontrar el subconjunto S de actividades compatibles de peso máximo (esto es, un conjunto de actividades que no se solapen en el tiempo y que, además, nos proporcione un valor máximo).

Programación Dinámica
Selección de actividades con pesos

Recordatorio

Existe un algoritmo greedy para este problema cuando todas las actividades tienen el mismo valor (elegir las actividades en orden creciente de hora de finalización).

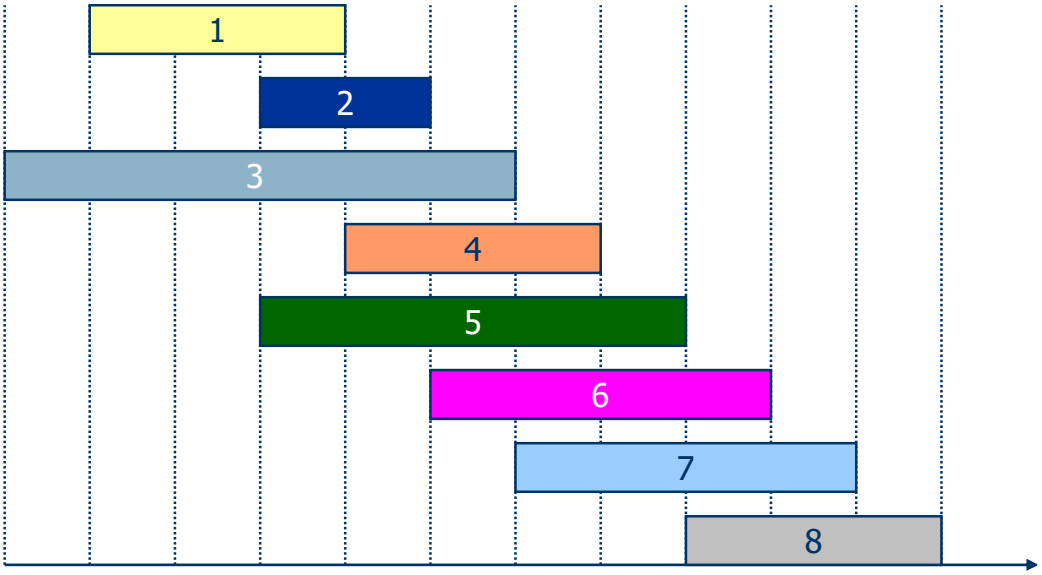
Sin embargo, el algoritmo greedy no funciona en general:



Programación Dinámica
Selección de actividades con pesos

Observación

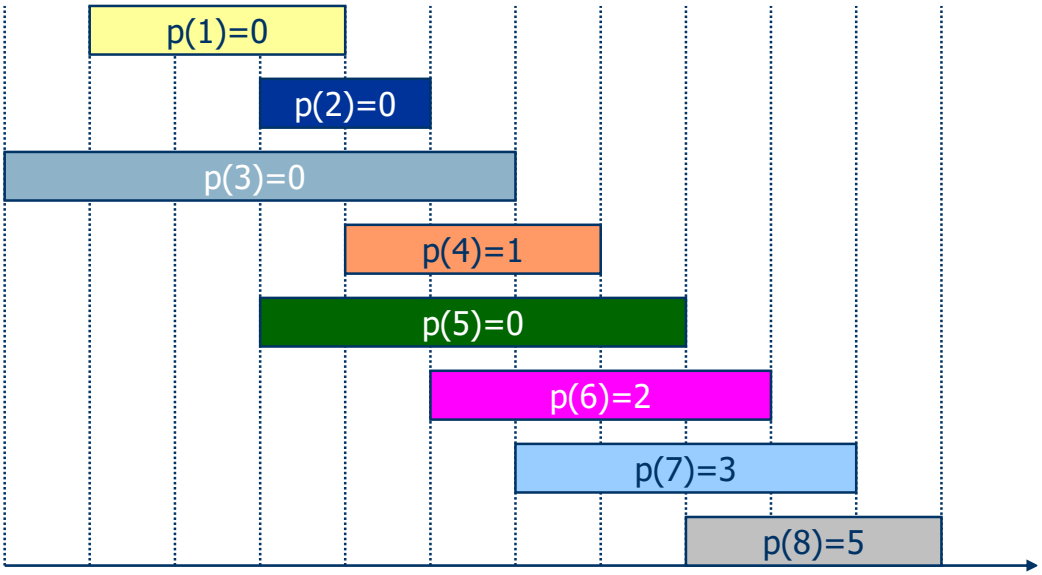
Si, como en el algoritmo greedy, ordenamos las actividades por su hora de finalización…



Programación Dinámica
Selección de actividades con pesos

Observación

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



Programación Dinámica
Selección de actividades con pesos

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: Se elige la actividad j.
  - No se pueden escoger actividades incompatibles >p(j).
  - La solución incluirá la solución óptima para p(j).
- Caso 2: No se elige la actividad j.
  - La solución coincidirá con la solución óptima para las primeras (j-1) actividades.

Programación Dinámica
Selección de actividades con pesos

Implementación iterativa del algoritmo

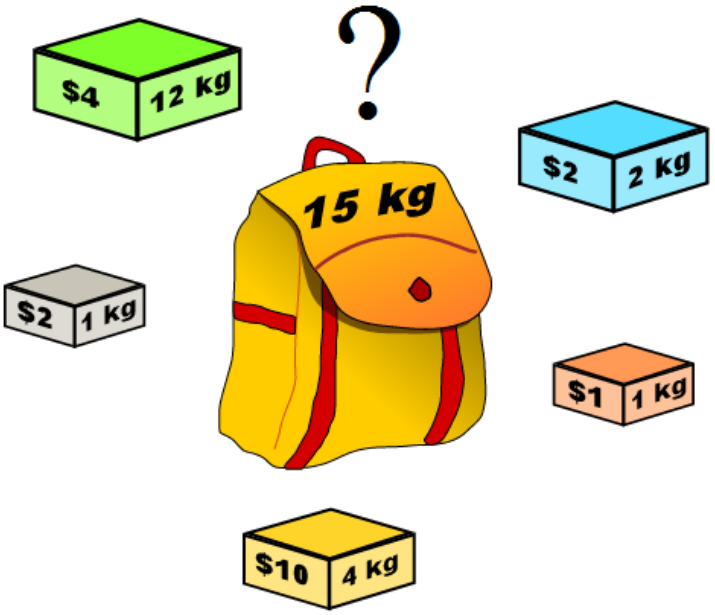
```
SelecciónActividadesConPesos (C: actividades): S
{
  Ordenar C según tiempo de finalización;        // O(n log n)
  Calcular p[1]..p[n];                           // O(n log n)

  mejor[0] = 0;                                  // O(1)
  for (i=1; i<=n, i++)                           // O(n)
      mejor[i] = max ( valor[i]+mejor[p[i]],
                       mejor[i-1] );

  return Solución(mejor);                        // O(n)
}
```

Programación Dinámica
El problema de la mochila 0/1

Tenemos un conjunto S de n objetos, en el que cada objeto i tiene un beneficio b_i y un peso w_i positivos.



Objetivo: Seleccionar los elementos que nos garantizan un beneficio máximo pero pero con un peso global menor o igual que W.

Programación Dinámica
El problema de la mochila 0/1

Dado el conjunto S de n objetos,
sea S_k el conjunto de los k primeros objetos (de 1 a k):

Podemos definir **B(k,w)** como la ganancia de la mejor solución obtenida a partir de los elementos de S_k para una mochila de capacidad w.

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

Programación Dinámica
El problema de la mochila 0/1

¿Cómo calculamos B(k,w)?

O bien la mejor opción para S_k coincide con la mejor selección de elementos de S_{k-1} con peso máximo w (el beneficio máximo para S_k coincide con el de S_{k-1}),

o bien es es el resultado de añadir el objeto k a la mejor selección de elementos de S_{k-1} con peso máximo $w-w_k$ (el beneficio para S_k será el beneficio que se obtenía en S_{k-1} para una mochila de capacidad $w-w_k$ más el beneficio b_k asociado al objeto k).

Programación Dinámica
El problema de la mochila 0/1

Definición recursiva de B(k,w):

$$B(k,w) = \begin{cases} B(k-1,w) & si \quad x_k = 0 \\ B(k-1,w-w_k)+b_k & si \quad x_k = 1 \end{cases}$$

Para resolver el problema de la mochila nos quedaremos con el máximo de ambos valores:

$$B(k,w) = \max\{B(k-1,w), B(k-1,w-w_k)+b_k\}$$

Programación Dinámica
El problema de la mochila 0/1

Cálculo ascendente de B(k,w)

usando una matriz B de tamaño (n+1) x (W+1):

```
int[][] knapsack (W, w[1..n], b[1..n])
{
  for (p=0; p<=W; p++)
      B[0][p]=0;

  for (k=1; k<=n; k++) {
      for (p=0; p<w[k]; p++)
          B[k][p] = B[k-1][p];
      for (p=w[k]; p<=W; p++)
          B[k][p] = max ( B[k-1][p-w[k]]+b[k], B[k-1][p] );
  }

  return B;
}
```

Programación Dinámica
El problema de la mochila 0/1

¿Cómo calculamos la solución óptima a partir de B(k,w)?

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

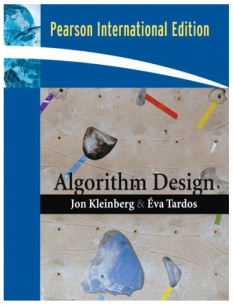
Si B[k][w] == B[k-1][w],
entonces el objeto k no se selecciona y se seleccionan los objetos correspondientes a la solución óptima para k-1 objetos y una mochila de capacidad w:
la solución para B[k-1][w].

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

Eficiencia del algoritmo

Tiempo de ejecución: **$\Theta(n\,W)$**

- "Pseudopolinómico" (no es polinómico sobre el tamaño de la entrada; esto es, sobre el número de objetos).
- El problema de la mochila es NP.



NOTA: Existe un algoritmo polinómico de orden $O(n^2v^*)$ que proporciona una solución aproximada al 0.01% del óptimo [Kleinberg & Tardos, sección 11.8: "Arbitrarily Good Approximations: The Knapsack Problem"].

Ejemplo

Mochila de tamaño **W=11**

Número de objetos **n=5**

Solución óptima **{3,4}**

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

	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

Programación Dinámica
Caminos mínimos: Algoritmo de Floyd

Problema:

Calcular el camino más corto que une cada par de vértices de un grafo, considerando que no hay pesos negativos.

Posibles soluciones:

- Por fuerza bruta (de orden exponencial).
- Aplicar el algoritmo de Dijkstra para cada vértice.
- Algoritmo de Floyd (programación dinámica).

Programación Dinámica
Caminos mínimos: Algoritmo de Floyd

Definición recursiva de la solución:

$D_k(i,j)$: Camino más corto de i a j usando sólo los k primeros vértices del grafo como puntos intermedios.

Expresión recursiva:

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

Caso base:

$$D_0(i,j) = c_{ij}$$

Programación Dinámica
Caminos mínimos: Algoritmo de Floyd

Algoritmo de Floyd (1962): $\Theta(V^3)$

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        D[i][j] = coste(i,j);

for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
               D[i][j] = D[i][k] + D[k][j];
```

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

Programación Dinámica
Caminos mínimos: Algoritmo de Floyd

Algoritmo de Floyd (1962): $\Theta(V^3)$


```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        D[i][j] = coste(i,j);
        P[i][j] = null;

for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (D[i][k] + D[k][j] < D[i][j] )
               D[i][j] = D[i][k] + D[k][j];
               P[i][j] = k;
```

Matriz `P` para reconstruir los caminos cómodamente…

Algoritmo de Floyd (1962):

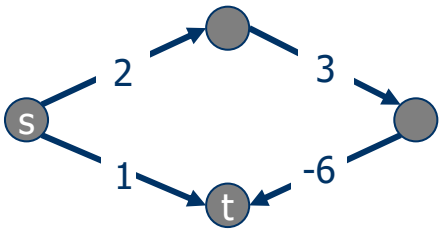
Reconstrucción de los caminos más cortos

```
path(P,i,j)      // Devuelve el camino más corto de i a j
{
  if (P[i][j]==null) {       // Conexión directa de i a j
     return { (i,j) };
  } else {                   // Camino pasando por k
     k = P[i][j];
     return path(P,i,k) U path(P,k,j);
  }
}
```

Reconstrucción recursiva de los caminos más cortos pasando por un nodo intermedio k entre i y j (i→k→j).

Si sólo nos interesan los caminos mínimos desde un vértice concreto del grafo G(V,E), podemos utilizar el algoritmo greedy de Dijkstra, de orden **O(E log V)**, siempre y cuando tengamos **pesos no negativos**.

El algoritmo de Dijkstra no funciona con pesos negativos:



¡Ojo!

Tampoco podemos sumarle una constante a cada peso.

Programación Dinámica
Caminos mínimos: Algoritmo de Bellman-Ford

Si tenemos pesos negativos, podemos utilizar el algoritmo de Bellman-Ford, basado en programación dinámica y de orden **O(EV)**, siempre y cuando no tengamos ciclos de peso negativo:

$$D_i(w)=\begin{cases}\infty & \text{si } i=0\\ \min\{D_{i-1}(w), \min\limits_{(v,w)\in E}\{D_{i-1}(v)+c_{vw}\}\} & \text{en otro caso}\end{cases}$$

NOTA: Si un camino de s a t incluye un ciclo de peso negativo, no existe un camino "más corto" de s a t (y, en cualquier caso, Bellman-Ford tampoco encontraría el camino simple más corto).

Programación Dinámica
Caminos mínimos: Algoritmo de Bellman-Ford

Algoritmo de Bellman-Ford: Θ(EV)

```
foreach v ∈ V {
      D[v] = ∞;
      predecesor[v] = null;
}
D[s] = 0;
for (i=1; i<n; i++) {
    foreach (v, w) ∈ E {
        if (D[v] + coste(v,w) < D[w]) {
           D[w] = D[v] + coste(v,w);
           predecesor[w] = v;
        }
    }
}
```

Algoritmo de Bellman-Ford:

Reconstrucción de los caminos más cortos

```
path (v)
{
  ruta = [v];
  w = v;
  while (predecesor[w]!=null) {
     w = predecesor[w];
     ruta = [w] U ruta;
  }
  return ruta;
}
```


También conocida como distancia Levenshtein, mide la diferencia entre dos cadenas s y t como el número mínimo de operaciones de edición que hay que realizar para convertir una cadena en otra:

d("data mining", "data minino") = 1
d("efecto", "defecto") = 1
d("poda", "boda") = 1
d("night","natch") = d("natch","noche") = 3

Aplicaciones: Correctores ortográficos, reconocimiento de voz, detección de plagios, análisis de ADN…

Para datos binarios: Distancia de Hamming.

Programación Dinámica
Distancia de edición

Definición recursiva de la solución

$$d(i,j) = \begin{cases} d(i-1,j-1) & si \quad s[i] = t[j] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & si \quad s[i] \neq t[j] \end{cases}$$

CASOS

- Mismo carácter: d(i-1,j-1)
- Inserción: 1 + d(i-1,j)
- Borrado: 1 + d(i,j-1)
- Modificación: 1 + d(i-1,j-1)

Programación Dinámica
Distancia de edición

```
int LevenshteinDistance (string s[1..m], string t[1..n])
{
  for (i=0; i<=m; i++) d[i,0]=i;
  for (j=0; j<=n; j++) d[0,j]=j;

  for (j=1; j<=n; j++)
      for (i=1; i<=m; i++)
          if (s[i]==t[j])
             d[i,j] = d[i-1, j-1]
          else
             d[i,j] = 1+ min(d[i-1,j],d[i,j-1],d[i-1,j-1]);

  return d[m,n];
}
```

$$d(i,j) = \begin{cases} d(i-1,j-1) & si \quad s[i] = t[j] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & si \quad s[i] \neq t[j] \end{cases}$$

Programación Dinámica
Aplicaciones

- Distancia entre dos series temporales: $O(n^2)$
  DTW [Dynamic Time Warping]
- Análisis sintáctico: $O(n^3)$
  - Algoritmo CKY [Cocke-Kasami-Younger]
  - Algoritmo de Earley
- Algoritmo de Viterbi: HMMs [Hidden Markov Models]
  - Decodificación de señales (p.ej. modems, GSM, wi-fi)
  - Procesamiento del lenguaje natural
  - Bioinformática
- Optimización de consultas en bases de datos relacionales
- Comparación de ficheros con *diff*
- …