

Algorítmica

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos “Divide y vencerás”

Tema 4. Algoritmos Voraces (“Greedy”)

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Algoritmos para la Exploración de Grafos
 (“Backtracking”, “Branch and Bound”)

Tema 7. Otras metodologías algorítmicas

Programación Dinámica

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

Introducción a la PD

- Esta técnica se aplica sobre problemas que a simple vista necesitan un alto coste computacional (posiblemente exponencial) donde:
 - **Subproblemas óptimos:** La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor, generalmente de forma recursiva.
 - **Solapamiento entre subproblemas:** Al plantear la solución recursiva, un mismo problema se resuelve más de una vez

Estrategias de diseño

Algoritmos greedy:

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

Divide y vencerás:

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

Programación dinámica:

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

Introducción a la PD

Se puede seguir un 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.

Introducción a la PD

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).

Introducción a la PD

- **Ejemplo. Cálculo de los números de Fibonacci.**

$$F(n) = \begin{cases} 1 & \text{Si } n \leq 2 \\ F(n-1) + F(n-2) & \text{Si } n > 2 \end{cases}$$

- **Con divide y vencerás.**

operación Fibonacci (n: entero): entero

si $n \leq 2$ entonces devolver 1

sino devolver Fibonacci(n-1) + Fibonacci(n-2)

- **Con programación dinámica.**

operación Fibonacci (n: entero): entero

$T[1] := 1; T[2] := 1$

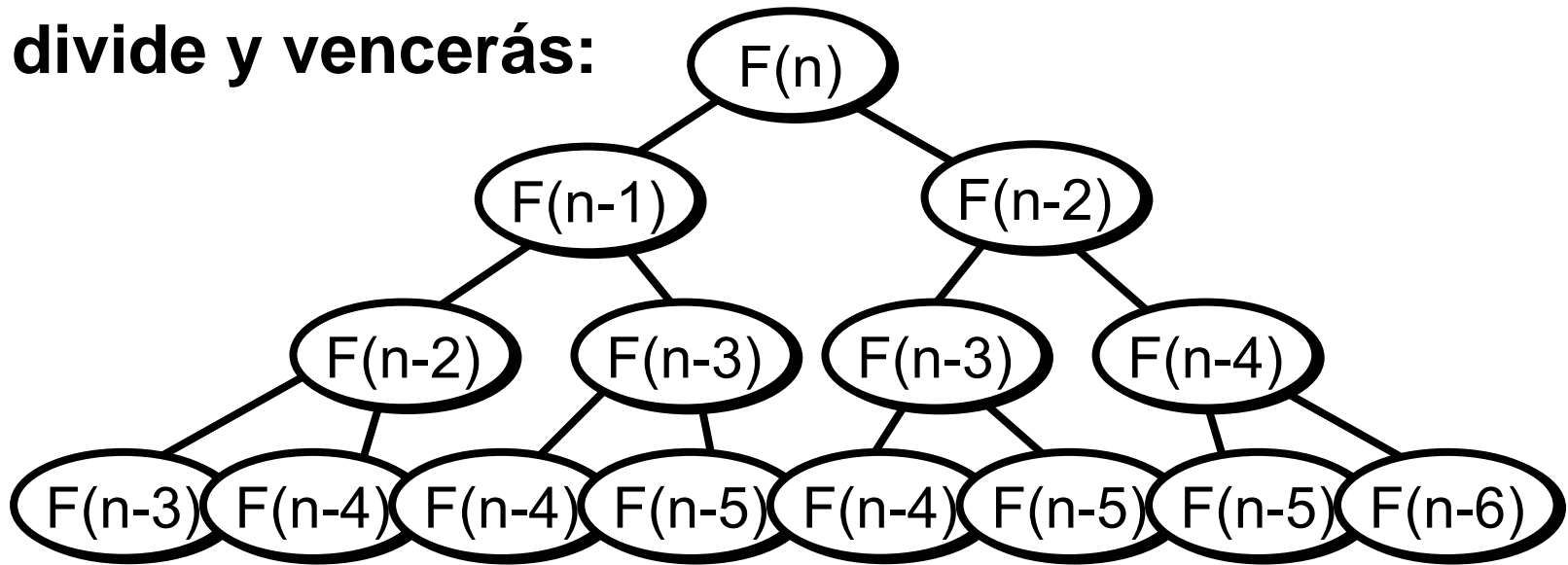
para $i := 3, \dots, n$ hacer

$T[i] := T[i-1] + T[i-2]$

devolver $T[n]$

Introducción a la PD

- Los dos usan la misma fórmula recursiva, aunque de forma distinta.
- ¿Cuál es más eficiente?
- **Con programación dinámica:** $\Theta(n)$
- **Con divide y vencerás:**



- **Problema:** Muchos cálculos están repetidos (Solapamiento de los subproblemas)
- El tiempo de ejecución es exponencial: $\Theta(1,62^n)$

Introducción a la PD

- La base de la **programación dinámica** es el **razonamiento inductivo**: ¿cómo resolver un problema combinando soluciones para problemas más pequeños?
- La idea es la misma que en **divide y vencerás**... pero aplicando una estrategia distinta.
- **Similitud**:
 - Descomposición recursiva del problema.
 - Se obtiene aplicando un razonamiento inductivo.
- **Diferencia**:
 - Divide y vencerás: aplicar directamente la fórmula recursiva (programa recursivo).
 - Programación dinámica: resolver primero los problemas más pequeños, guardando los resultados en una tabla (programa iterativo).

Introducción a la PD

Métodos ascendentes y descendentes

- **Métodos descendentes (divide y vencerás)**
 - Empezar con el problema original y descomponer recursivamente en problemas de menor tamaño.
 - Partiendo del problema grande, descendemos hacia problemas más sencillos.
- **Métodos ascendentes (programación dinámica)**
 - Resolvemos primero los problemas pequeños (guardando las soluciones en una tabla). Después los vamos combinando para resolver los problemas más grandes.
 - Partiendo de los problemas pequeños avanzamos hacia los más grandes.

Introducción a la PD

Pasos para aplicar programación dinámica:

- 1) Obtener una **descomposición recurrente** del problema:
 - Ecuación recurrente.
 - Casos base.
 - 2) Definir la **estrategia** de aplicación de la fórmula:
 - Tablas utilizadas por el algoritmo.
 - Orden y forma de rellenarlas.
 - 3) Especificar cómo se **recompone la solución** final a partir de los valores de las tablas.
- **Punto clave:** obtener la descomposición recurrente.
 - Requiere mucha “creatividad”...

Introducción a la PD

- Cuestiones a resolver en el 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”)?
- **Idea:** ver lo que ocurre al tomar una decisión concreta
→ interpretar el problema como un proceso de toma de decisiones.
- **Ejemplo: Mochila 0/1.** Decisiones: coger o no coger un objeto dado.

Introducción a la PD

- La programación dinámica se basa en el uso de **tablas** donde se almacenan los resultados parciales.
- En general, el **tiempo** será de la forma:
Tamaño de la tabla * Tiempo de rellenar cada elemento de la tabla.
- Un aspecto **importante** es la memoria puede llegar a ocupar la tabla.
- Además, algunos de estos cálculos pueden ser innecesarios.

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.

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 $e_1e_2\dots e_k\dots e_n$ entonces:

- $e_1e_2\dots e_k$ es solución al problema P considerando los k primeros elementos.
- $e_{k+1}\dots e_n$ 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

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”.

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 podemos 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.

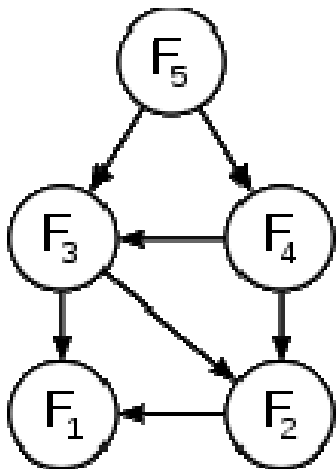
Programación Dinámica

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

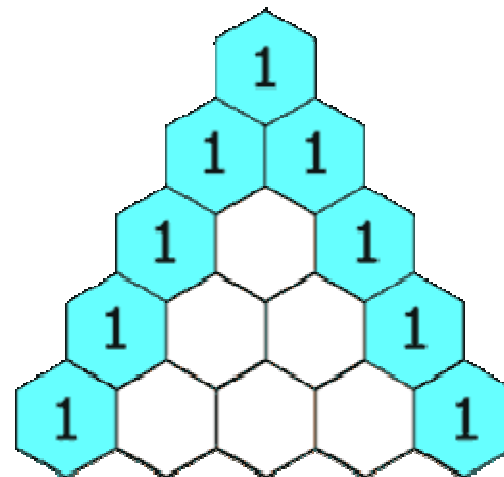
Ejemplos

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

(n elementos tomados de p en 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: $\Theta(np)$ en tiempo, $\Theta(np)$ en espacio.

Programación Dinámica

Ejemplos

Números combinatorios:

Combinaciones de n sobre 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];
}
```

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

Orden de eficiencia: $\Theta(np)$ en tiempo, $\Theta(n)$ 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 (usando división entera)

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

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...

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

	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

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:

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

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:

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

$$\boxed{A_1 \times A_2 \times A_3 \times \dots \times A_k} \times \boxed{A_{k+1} \times A_{k+2} \times \dots \times A_n}$$

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 $i \neq j$, entonces

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

$A_1 \times A_2 \times A_3 \times \dots \times A_k$	\times	$A_{k+1} \times A_{k+2} \times \dots \times A_n$
---	----------	--

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

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

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

Programación Dinámica

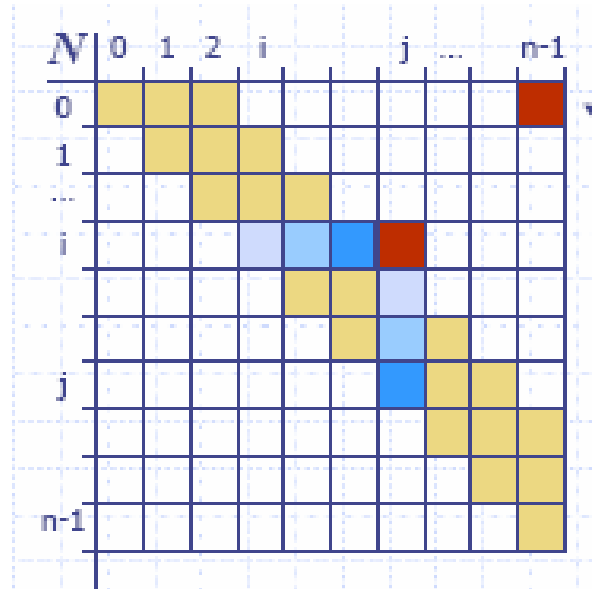
Implementación (Índices de 1 a n):

```

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

for (s=2; s<=n; s++)
    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:

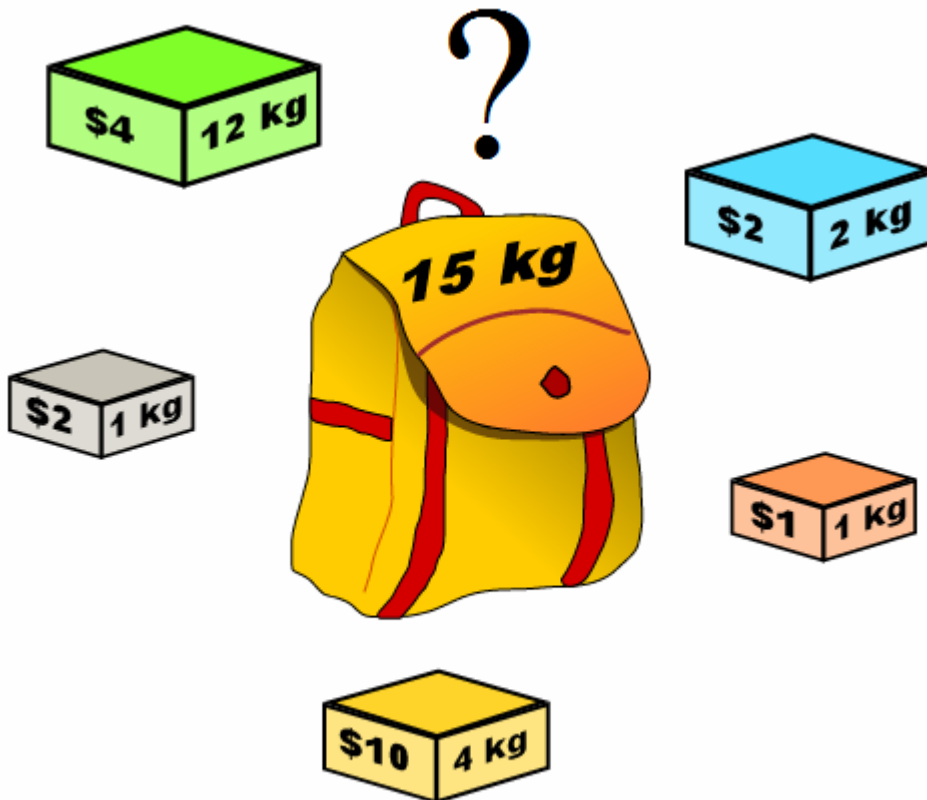
// Suponiendo que hemos calculado previamente $s[i,j]$...

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

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 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 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 $B(k, w)$:

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

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 $(n+1) \times (W+1)$:

```
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 $B[k-1][w-w[k]]$.

Programación Dinámica

El problema de la mochila 0/1

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.

Programación Dinámica

El problema de la mochila 0/1

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	34	40

Subsecuencia Común de Mayor Longitud (LCS)

Aplicacion: comparar dos cadenas de DNA

Ej: $X = \{A B C B D A B\}$, $Y = \{B D C A B A\}$

Subsec. Común de Mayor longitud :

$X = A \text{ **B** } \text{ **C** } \text{ **B** } D \text{ **A** } B$

$Y = \text{ **B** } D \text{ **C** } A \text{ **B** } \text{ **A** }$

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

Algoritmo LCS

- Si $|X| = m$, $|Y| = n$, entonces hay 2^m subsecuencias de x ; y las debemos comparar con Y (n comparaciones) $\rightarrow O(n 2^m)$
- Sin embargo, LCS tiene *subestructuras optimales*: las soluciones a los subproblemas son parte de la solución final.
- Subproblemas: “Encontrar LCS para pares de prefijos de X e Y ”

Algoritmo LCS

- Definimos X_i, Y_j los prefijos de X e Y de longitud i y j respectivamente
- Definimos $c[i,j]$ la longitud de LCS para X_i e Y_j
- Entonces, LCS de X e Y será $c[m,n]$

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

Definición Recursiva

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

- *Inicio:* $i = j = 0$ (subcadena vacía de x e y)

$$(c[0, 0] = 0)$$

- LCS de la cadena vacía y cualquier otra cadena es vacía, por tanto para cada par i, j :

$$c[0, j] = c[i, 0] = 0$$

Definición Recursiva

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

- Cuando calculamos $c[i, j]$, consideramos dos casos:
- **Primer caso:** $x[i] = y[j]$: Se emparejan un símbolo más en X e Y.
- **Segundo caso:** $x[i] \neq y[j]$: Como no se emparejan los símbolos, la longitud no se mejora y es la mejor entre $LCS(X_i, Y_{j-1})$ y $LCS(X_{i-1}, Y_j)$

Algoritmo LCS

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // get the # of symbols in Y
3. for $i = 0$ to m $c[i,0] = 0$ // special case: $c_{0,0} \in Y_0$
4. for $j = 1$ to n $c[0,j] = 0$ // special case: X_0
5. for $i = 1$ to m // for all X_i
6. for $j = 1$ to n // for all Y_j
7. if ($X_i == Y_j$)
8. $c[i,j] = c[i-1,j-1] + 1$
9. else $c[i,j] = \max(c[i-1,j], c[i,j-1])$
10. return c

Ejemplo LCS

- $X = \text{ABCB}$
- $Y = \text{BDCAB}$

$\text{LCS}(X, Y) = \text{BCB}$

$X = \text{A } \mathbf{B} \quad \mathbf{C} \quad \mathbf{B}$

$Y = \quad \mathbf{B} \text{ D } \mathbf{C} \text{ A } \mathbf{B}$

Ejemplo LCS (0)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i								
0	X _i							
1	A							
2	B							
3	C							
4	B							

$X = ABCB; \quad m = |X| = 4$

$Y = BDCAB; \quad n = |Y| = 5$

Ejemplo LCS (1)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

Ejemplo LCS (2)

ABCB
BDCAB

		j	0	1	2	3	4	5
		Y _j		B	D	C	A	B
i	X _i							
0		0	0	0	0	0	0	0
1	A	0	0	0				
2	B	0						
3	C	0						
4	B	0						

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (3)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0		
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (4)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (5)

ABCB
 BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (6)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (7)

ABCB
BDCA

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0	0	0
1	A	0	0	0	0	0	1	1
2	B	0	1	1	1	1		
3	C	0						
4	B	0						

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (8)

ABCB
BDCA

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (10)

ABCB
BD CAB

i	j	0	1	2	3	4	5
		Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (11)

ABCB
BD CAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (13)

ABCB
BDCAB

		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
i	Xi							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (14)

ABCB
BDCA

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i								
0	X _i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

Ejemplo LCS (15)

ABCB
BDCA

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i		X _i						
0			0	0	0	0	0	0
1		A	0	0	0	0	1	1
2		B	0	1	1	1	1	2
3		C	0	1	1	2	2	2
4		B	0	1	1	2	2	3

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$


Cómo encontrar la subsecuencia LCS

Cada $c[i,j]$ depende de $c[i-1,j]$ y $c[i,j-1]$

O bien $c[i-1,j-1]$

Por tanto, a partir del valor $c[i,j]$ podremos averiguar cómo se determinó

2	2
2	3



Por ejemplo

$$c[i,j] = c[i-1,j-1] + 1 = 2+1=3$$

Cómo encontrar la subsecuencia LCS

- Como

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- Empezamos desde $c[m, n]$ y vamos hacia atrás
- Si $c[i, j] = c[i-1, j-1] + 1$ y $x[i] = x[j]$, guardamos $x[i]$ (porque $x[i]$ pertenece a LCS)
- En otro caso seguimos por $\max(c[i, j-1], c[i-1, j])$
- Si $i=0$ o $j=0$ (alcanzamos el principio), devuelve-mos los caracteres almacenados en orden inverso.

Encontramos LCS

		j	0	1	2	3	4	5
i			Y _j	B	D	C	A	B
		X _i						
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

Encontramos LCS

		j	0	1	2	3	4	5
i		Yj		B	D	C	A	B
	Xi							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

LCS :

B C B

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).

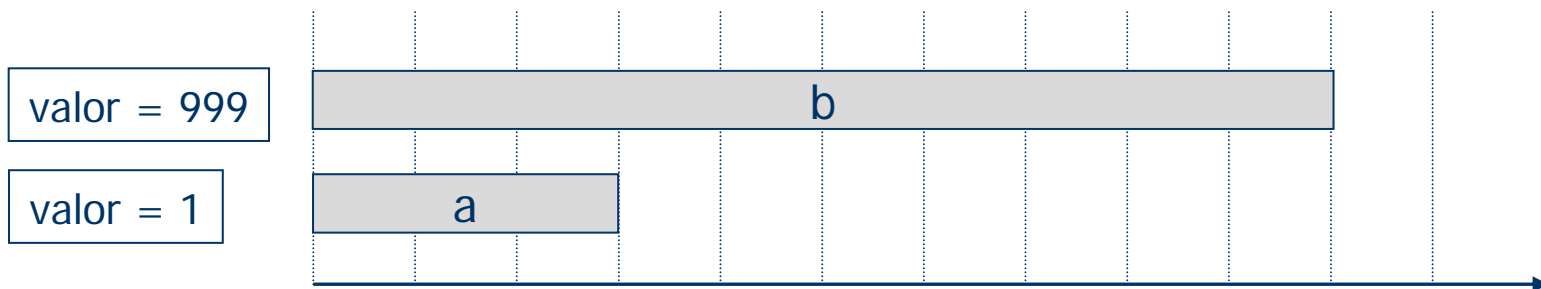
Programación Dinámica

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:

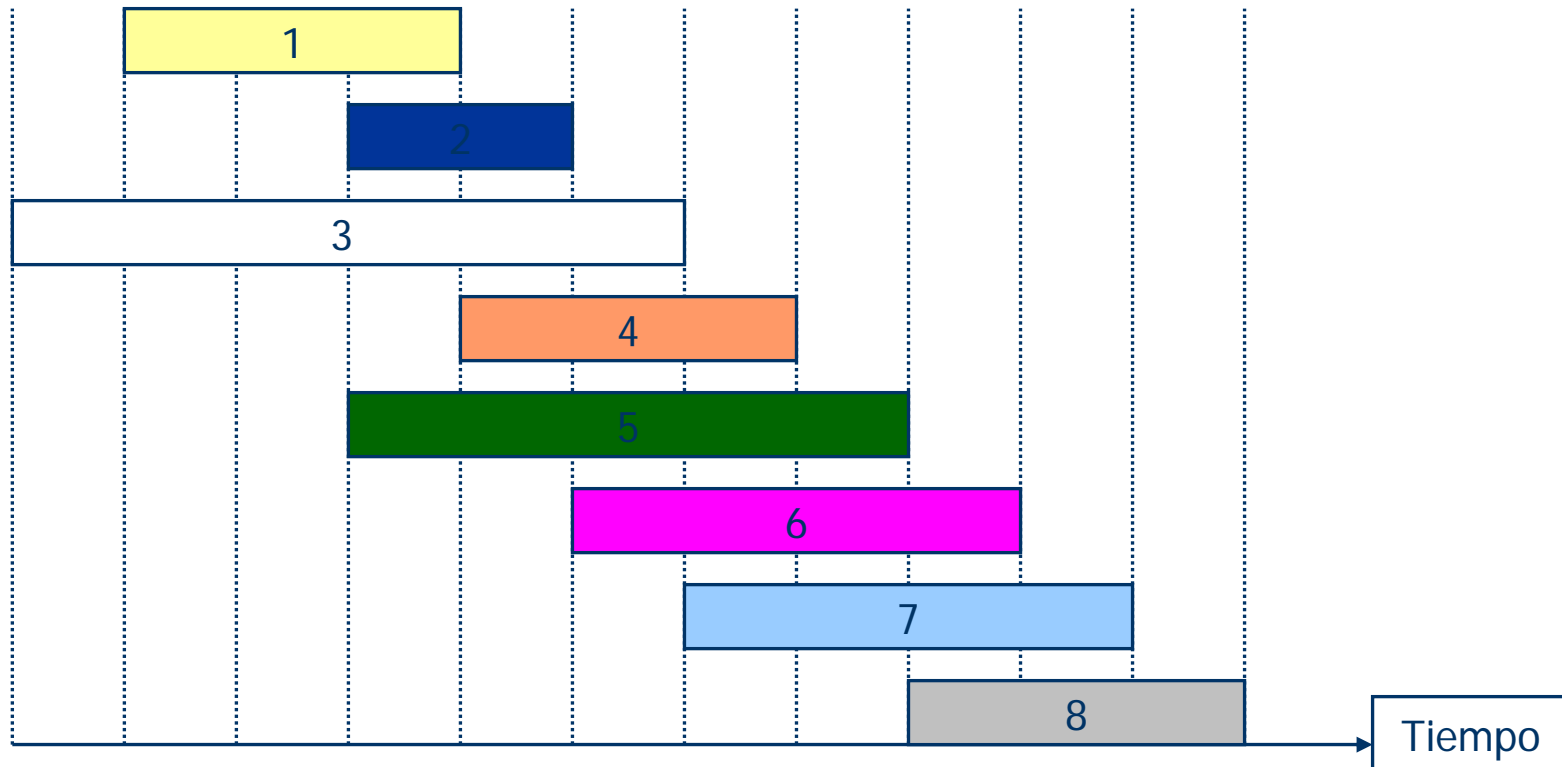


Programación Dinámica

Selección de actividades con pesos

Observación

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

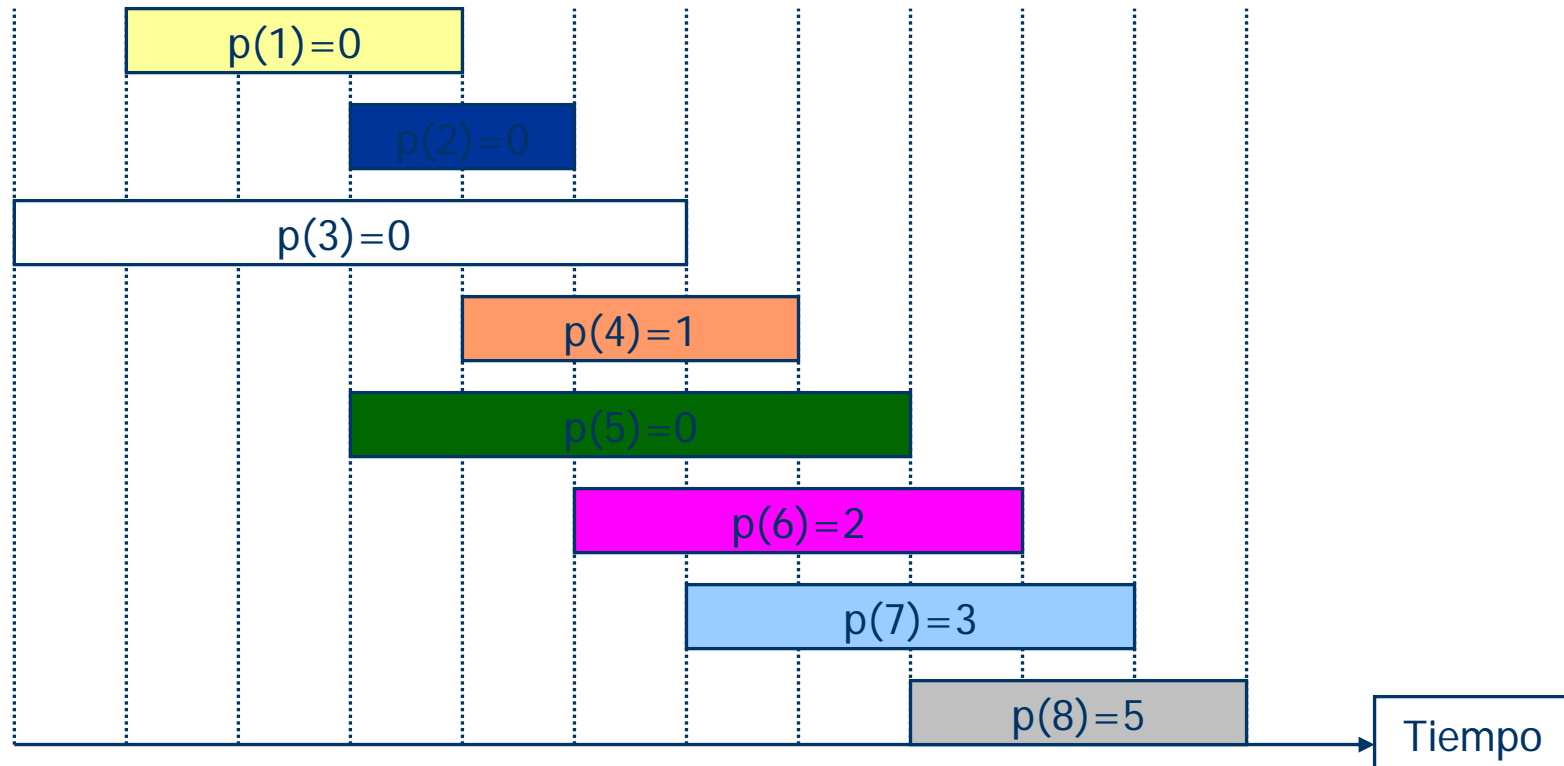


Programación Dinámica

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

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

- 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)
}
```

Nota: Faltaría el algoritmo "Solución()" que determinase cuál es la solución óptima para una lista de actividades. Dicho algoritmo chequearía los valores almacenados en "mejor" de manera lineal. Dicho algoritmo queda como ejercicio.

Programación Dinámica

Distancia de edición

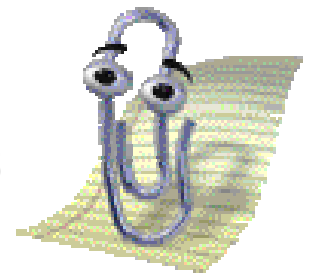
También conocida como distancia Levenshtein, mide la diferencia entre dos cadenas s y t como el número mínimo de operaciones de edición que hay que realizar para convertir una cadena en otra:

$d(\text{"data mining"}, \text{"data minino"}) = 1$

$d(\text{"efecto"}, \text{"defecto"}) = 1$

$d(\text{"poda"}, \text{"boda"}) = 1$

$d(\text{"night"}, \text{"natch"}) = d(\text{"natch"}, \text{"noche"}) = 3$



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

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

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];
}
```

Nota: Faltaría el algoritmo que determinase cuál es la solución óptima para dos cadenas dadas. Dicho algoritmo puede encontrarse en los ejercicios de PD resueltos (material adicional).

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

Programación Dinámica

Camino 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

Camino 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

Camino 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];
```

Nota: Faltaría el algoritmo que determinase cuál es la solución óptima para un origen y destino dados. Dicho algoritmo puede encontrarse en los ejercicios de PD resueltos (material adicional).

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

Programación Dinámica

Camino mínimos: Floyd Vs. Dijkstra

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

En caso de necesitar todos los caminos, Floyd es siempre $\Theta(V^3)$ mientras Dijkstra necesita ser ejecutado V veces, siendo $O(VA \log V)$. Si el grafo es muy denso $A \approx V^2$, mientras que si es poco denso $A \approx V$. El mejor algoritmo depende por tanto de la densidad del grafo.

Programación dinámica.

Conclusiones

- El **razonamiento inductivo** es una herramienta muy potente en resolución de problemas.
- Aplicable no sólo en problemas de optimización.
- ¿Cómo obtener la fórmula? Interpretar el problema como una serie de **toma de decisiones**.
- Descomposición recursiva no necesariamente implica implementación recursiva.
- **Programación dinámica:** almacenar los resultados en una tabla, empezando por los tamaños pequeños y avanzando hacia los más grandes.

Algorítmica

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos “Divide y vencerás”

Tema 4. Algoritmos Voraces (“Greedy”)

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Algoritmos para la Exploración de Grafos
 (“Backtracking”, “Branch and Bound”)

Tema 7. Otras metodologías algorítmicas