Programación Dinámica

Esquema

1. Motivación y características
2. Elementos de Programación Dinámica
   - Principio de Optimalidad de Bellman
   - Definición Recursiva de la solución optimal
   - Enfoque ascendente
   - Búsqueda solución óptima
3. Resolución de problemas tipo
   - Devolver cambio
   - Mochila 0/1
   - Multiplicación de Matrices
   - Subsecuencia de longitud mayor (LCS)
   - Caminos mínimos

Motivación: Fibonacci

- La sucesión de Fibonacci es:
- $F_n = n$ si $n=0$ o $n=1$
- $F_n = F_{n-1}+F_{n-2}$ si $n>1$
- Posible algoritmo de cálculo (recursivo)

```
función FIBONACCI ( n )
  si ( n <= 1 )
      devolver n
  si ( n > 1 )
      devolver FIBONACCI (n-1)+FIBONACCI (n-2)
```

Fibonacci

- Problema: no es eficiente porque se repiten muchos cálculos
- f6 = f5+f4 = f4+f3+f3+f2 = f3+f2+f2+f1+f2+f1+f2 = f2+f1+f2+f2+f1+f2+f1+f2
- f2 se calcula 5 veces!
- Los subproblemas se solapan
- Podríamos mejorar si almacenamos los resultados y calculamos cada fi una sola vez

```
fibDPwrap(n)
      Dict  soln = create(n);
      return fibDP(soln, n);

fibDP(soln, k)
      int  fib, f1, f2;
      if (k < 2)
            fib = k;
      else
            if (member(soln, k-1) == false)
                  f1 = fibDP(soln, k-1);
            else
                  f1 = retrieve(soln, k-1);

            if (member(soln, k-2) == false)
                  f2 = fibDP(soln, k-2);
            else
                  f2 = retrieve(soln, k-2);

            fib = f1 + f2;
      store(soln, k, fib);
      return fib;
```

Fibonacci lineal

- Podemos construir un algoritmo lineal usando una tabla para almacenar los cálculos y organizando los cálculos de forma ordenada:

```
If n<=1 return n;
Else {
        T[0]=0; T[1]=1;
        For i=2 to n
           T[i]=T[i-1]+T[i-2];
      }
      return T[n];
```

Fibonacci aun mejor

- Podemos hacer el algoritmo más eficiente en espacio, ahorrándonos la tabla:

```
If n<=1 return n;
Else {
        x=1; y=0;
        For i=2 to n
           suma=x+y;
           y=x;
           x=suma;
       }
       return suma;
```

Números combinatorios

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

calcula el número de subconjuntos de tamaño k que se pueden formar con n elementos distintos.

Se pueden calcular recursivamente

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

```
funcion C(n,k)
  si k=0 o k=n devolver 1
  sino devolver C(n-1,k-1)+C(n-1,k)
```

Números combinatorios

- Problema: Muchos de los valores C(i,j), con i<n,j<k se calculan varias veces
- C(5,3)=C(4,2)+C(4,3)=(C(3,1)+C(3,2))+(C(3,2)+C(3,3))=

  =C(2,0)+C(2,1)+C(2,1)+C(2,2)+C(2,1)+C(2,2)+1=
- =1+C(1,0)+C(1,1)+C(1,0)+C(1,1)+1+C(1,0)+C(1,1)+1+1=10
- Recurrencia f(n)=f(n-1)+f(n-2): complejidad exponencial

Números combinatorios

- Si utilizáramos una tabla de resultados intermedios C[i,j], i=1..n, j=0..k (el triángulo de Pascal) obtenemos un resultado más eficiente.
- Rellenando la tabla línea por línea:

	0	1	2	3	4	5	⋯
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
⋮			⋮				⋱

```
for i=1 to n
 for j=0 to k
  if j<=i
   if (j=0 or j=i) C[i,j]=1
   else C[i,j]=C[i-1,j-1]+C[i-1,j]
return C[n,k]
```

Números combinatorios

- Incluso podemos usar simplemente dos vectores de tamaño k (que representan la línea actual y la anterior) (o incluso un solo vector, actualizándolo de dcha. a izqda.)

- Este algoritmo requiere un tiempo O(nk) y un espacio O(k)

Motivación: Qué hemos visto

- Cuando un problema se divide recursivamente en subproblemas que se solapan, la eficiencia disminuye mucho.

- Se pueden mantener en memoria los subcasos resueltos para no repetir cálculos y mejorar la eficiencia.

Programación Dinámica: Características de los problemas

- Suelen ser problemas de optimización
- el problema debe poder resolverse por etapas, Sol=d1,d2,d3,...,dn, se toma una decisión en cada paso, pero esta depende de las soluciones a los subproblemas que lo componen
- debe poder modelizarse con una función recurrente
- debe cumplir el Principio de Optimalidad de Bellman

Programación Dinámica: Características de los problemas

- Esta técnica se aplica sobre problemas que a simple vista necesitan un alto coste computacional (posiblemente exponencial) donde:
  - **Subproblemas optimales**: 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.

PD: Características de la técnica

- Introducida por Richard Bellman en 1957
- No tiene nada que ver con la programación (es más bien “planificación”)
- Resuelve un problema por etapas (como greedy o backtracking)
- Divide el problema en subproblemas (como Divide y Vencerás)

PD: Características de la técnica

- Suele ser una técnica ascendente (bottom-up) para obtener la solución, primero calcula las soluciones óptimas a problemas de tamaño pequeño. Utilizando dichas soluciones encuentra soluciones a problemas de mayor tamaño.

- Retiene en memoria las soluciones de los subproblemas, para evitar cálculos repetidos (memoizing)

- Devuelve la solución óptima (principio de Optimalidad de Bellman)

PD: Idea general

- **Encontrar una formulación recursiva de la solución de un problema mayor en función de soluciones a problemas menores.**
- **Resolver cada instancia de tamaño menor una única vez y guardarla en una tabla.**
- **Obtener la solución de la instancia inicial utilizando las soluciones almacenadas.**

PD: Análisis de la eficiencia

- Depende del problema, aunque suele ser polinomial

- En general, al utilizar una tabla, será $n \cdot m$
  - n es el tamaño de la tabla
  - m tiempo para rellenar cada casilla

- Algunos cálculos pueden ser innecesarios
- Puede dar problemas al necesitar mucha memoria (para rellenar la tabla)

PD versus Divide y Vencerás

Ambos combinan soluciones de subproblemas, pero

- DyV se aplica cuando los subproblemas son independientes
- PD se aplica cuando los subproblemas se solapan
- DyV repetiría muchos cálculos. PD los mantiene en memoria
- DyV utiliza un método descendente
- PD utiliza un método ascendente (normalmente)
- DyV utiliza recurrencias (+ tiempo, - memoria)
- PD intenta evitar recurrencias, utiliza memoria para iteración (- tiempo, + memoria)
- En ambos casos se obtiene la solución óptima

PD versus Greedy

Ambos resuelven el problema por etapas, pero

- Greedy selecciona un elemento y sólo genera una solución, en cada etapa
- PD selecciona un elemento en cada paso, pero genera múltiples caminos de etapas a seguir. Elige la optimal entre ellas
- Greedy no asegura optimalidad (miope)
- PD asegura optimalidad (Principio de Optimalidad de
- Bellman)
- Greedy es eficiente en tiempo y en memoria
- PD es eficiente en tiempo pero no en memoria

Principio de Optimalidad de Bellman

Una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones debe ser también óptima respecto al subproblema que resuelve.

Esto es,

La solución óptima a cualquier caso no trivial de un problema es una combinación de soluciones óptimas de algunos de los subcasos.

Si d1,d2,d3, …, dn es optimal para P[1,n]

- *Entonces d1,d2,…,di es optimal para P[1,i]*
- *y di+1,…,dn es optimal para P[i+1,n]*

Principio de optimalidad

- La dificultad en aplicar este principio está en que no suele ser evidente cuáles son los subcasos relevantes para el caso considerado.
- Esto impide usar una aproximación similar a divide y Vencerás, comenzando en el caso original y buscando recursivamente soluciones óptimas para los subcasos relevantes y sólo para estos.
- En su lugar la PD resuelve todos los subcasos, para determinar los que realmente son relevantes; y entonces se combinan en una solución óptima para el caso original.

Pasos para desarrollar un algoritmo basado en PD

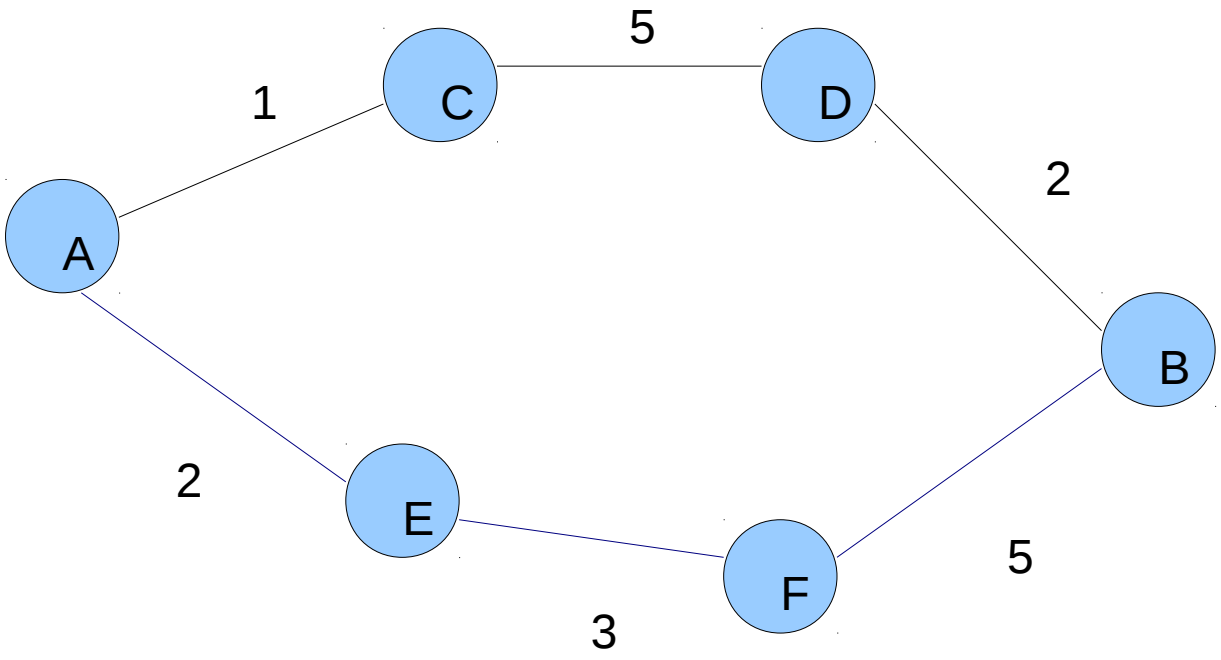
1. Planteamiento de la solución como una sucesión de decisiones y verificación de que se cumple el Principio de Optimalidad de Bellman:
   - Encontrar la estructura de la solución:
     - Dividir el problema en subproblemas y determinar si se puede aplicar el principio de optimalidad.
2. Definición recursiva de la solución optimal:
   - Definir el valor de la solución óptima en función de valores de soluciones para sub-problemas de tamaño menor.

Pasos para desarrollar un algoritmo basado en PD

3) Calcular el valor de la solución optimal utilizando un enfoque ascendente.
   - Determinar el conjunto de subproblemas distintos a resolver (tamaño de la tabla)
   - Identificar los subproblemas con solucion trivial
   - Obtener los valores con un enfoque ascendente y almacenar los valores que vamos calculado en la tabla.
   - En etapas posteriores se utilizaran los valores previamente calculados
4) Determinar la solución óptima a partir de la información préviamente calculada.

Ejemplo de incumplimiento del POB

- Cálculo del camino simple (sin ciclos) más largo entre dos nodos de un grafo



Ejemplo: Devolver cambio

- Compramos un artículo en una tienda
- el artículo cuesta 5.37 €
- pagamos con un billete de 10 €
- entonces nos tienen que devolver 4.63 €
- ¿Cómo nos devuelve el cambio con menor número de monedas?

Devolver cambio

El algoritmo Greedy era eficiente pero no eficaz:

- Si hay monedas de 1, 4 y 6 céntimos y hay que devolver 8 céntimos:
- el algoritmo greedy devuelve
  - 1 moneda de 6 céntimos
  - 2 monedas de 1 céntimo
- la solución óptima es 2 monedas de 4 céntimos
- Lo mejoramos con Programación Dinámica

Devolver cambio

El problema general es:

- monedas de n valores diferentes
- las monedas de tipo i tienen un valor de c_i>0 unidades
- hay un suministro ilimitado de monedas
- al cliente hay que devolverle un valor M
- hay que utilizar el menor número de monedas

Devolver cambio: POB?

- Se puede plantear la solución como una secuencia de decisiones x1,x2,...,xn: cuántas monedas de tipo 1, cuántas de tipo 2,...
- ¿Se cumple el POB?
- Sea m(i,j) el mínimo número de monedas para devolver una cantidad j usando solamente monedas de los tipos 1,2,...,i
- El problema original es m(n,M)

Demostración del POB

Si $x_1..x_n$ es óptima para m(n,M) entonces hay que demostrar que $x_1..x_{n-1}$ es óptima para m(n-1,M-x_n*c_n).

- Por contradicción
- Si no fuese así entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i < sum_1^{n-1} x_i y sum_1^{n-1} y_i*c_i = M-x_n*c_n
- Pero entonces, haciendo y_n=x_n tenemos que

  sum_1^{n} y_i*c_i = M, luego $y_1..y_n$ es una solución para m(n,M), y además

  sum_1^{n} y_i < sum_1^{n} x_i, luego $x_1..x_n$ no sería óptima

Definición recursiva de la solución óptima

- Si la solución para m(i,j) no incluye una moneda de tipo i, entonces

  $m(i,j)=m(i-1,j)$
- Si la solución para m(i,j) sí incluye una moneda de tipo i, entonces

  $m(i,j)=1+m(i,j-c_i)$
- Luego $m(i,j)=\min(m(i-1,j),1+m(i,j-c_i))$

Definición recursiva

- Más exactamente
  - m(i,j)=0 si j=0
  - m(i,j)=infinito si i=1 y 1<=j<c_i
  - m(i,j)=1+m(i,j-c_i) si i=1 y j>=c_i
  - m(i,j)=m(i-1,j) si i>1 y j<c_i
  - m(i,j)=min(m(i-1,j),1+m(i,j-c_i)) en otro caso
  - Necesitamos una tabla para almacenar los resultados de todos los subproblemas
  - Rellenamos la tabla de arriba a abajo y de izquierda a derecha

Devolver cambio: Algoritmo PD

```
For i=1 to n
   m[i,0]=0;
For i=1 to n
   For j=1 to M
      If (i==1 && j<c[i]) m[i,j]=10e30;
      Else if (i==1) m[i,j]=1+m[i,j-c[1]];
              Else if (j<c[i]) m[i,j]=m[i-1,j];
                      Else m[i,j]=min(m[i-1,j],1+m[1,j-c[i]]);
Return m[n,M];
```

Donde i==1 no se mantendría el

La eficiencia del algoritmo es O(nM)

Ejemplo

m[i,j]=min(m[i-1,j],1+m[i,j-c[i]])

n= 3, M= 8, c= (1, 4, 6)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1 = 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

Esta tabla solo da el número mínimo de monedas necesario para cada valor de M.

¿Cómo calcular cuántas monedas de cada tipo hacen falta?

Cuántas monedas de cada tipo?

- Empezamos con m[n,M]: i=n; j=M;
- Si m[i,j] = m[i-1,j] no se escoge una moneda de tipo i y pasamos a estudiar m[i-1,j]: i=i-1;
- Si m[i,j] = 1+m[1,j-c[i]] se escoge una moneda de tipo i y pasamos a estudiar m[i,j-c[i]]: j=j-c[i]; monedas[i]++;
- Hasta que lleguemos a algún m[i,0], y ya no quede nada por pagar

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C_1 = 1	0	1	2	3	4	5	6	7	8
C_2 = 4	0	1	2	3	1	2	3	4	2
C_3 = 6	0	1	2	3	1	2	1	2	2

Ejercicio: adaptar el algoritmo de PD al caso en que hay un número limitado de monedas de cada tipo

Mochila 0/1

Tenemos un conjunto S de n objetos, donde cada objeto, i, tiene

- b_i – beneficio positivo
- w_i – peso positivo

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

- Los objetos no se pueden partir

Mochila 0/1: POB?

- Se puede plantear la solución como una secuencia de decisiones x1,x2,...,xn: seleccionar o no el objeto 1, seleccionar o no el objeto 2,...
- ¿Se cumple el POB?
- Sea B(k,w) el mejor valor obtenido para un peso máximo de w usando solamente los objetos 1,2,...,k
- La solución del problema original es B(n,W)

Demostración del POB

Si $x_1..x_n$ es óptima para B(n,W) entonces hay que demostrar (por contradicción) que $x_1..x_{n-1}$ es óptima para:

- B(n-1,W) si $x_n=0$ ó B(n-1,W-w_n) si $x_n=1$
- Caso $x_n=0$. Si no fuese así entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i*b_i > sum_1^{n-1} x_i*b_i y sum_1^{n-1} y_i*w_i <=W
- Pero entonces, haciendo $y_n=x_n=0$ tenemos que

  sum_1^{n} y_i*w_i <= W, luego $y_1..y_n$ es una solución para B(n,W), y además

  sum_1^{n} y_i*b_i > sum_1^{n} x_i*b_i, luego $x_1..x_n$ no sería óptima

Demostración del POB

Caso $x_n=1$

- Si $x_1..x_{n-1}$ no es óptima para $B(n-1,W-w_n)$ entonces existe $y_1..y_{n-1}$ tal que

  sum_1^{n-1} y_i*b_i > sum_1^{n-1} x_i*b_i y

  sum_1^{n-1} y_i*w_i <=$W-w_n$
- Pero entonces, haciendo $y_n=x_n=1$ tenemos que

  sum_1^{n-1} $y_i*w_i+w_n$ <= W, luego $y_1..y_n$ es una solución para B(n,W), y además

  sum_1^{n-1} $y_i*b_i+b_n$> sum_1^{n-1} $x_i*b_i+b_n$, luego $x_1..x_n$ no sería óptima

Princ. de Optimalidad de Bellman

- La mejor selección de elementos del conjunto 1,2,...,k para una mochila de tamaño w se puede definir en función de selecciones de elementos de 1,2,...,k-1, para mochilas de tamaño menor.

- O bien

  es la ganancia para la mejor selección de elementos de 1,2,...,k-1 con peso máximo w
- O bien

  es la ganancia de la mejor selección de elementos de 1,2,...,k-1 con peso máximo $w-w_k$ más la ganancia del elemento k, b_k.

Definición Recursiva

$$B[k,w] = \begin{cases} B[k-1,w] & si \quad w_k > w \\ \max\left(B[k-1,w], B[k-1,w-w_k] + b_k\right) & en\ otro\ caso \end{cases}$$

Caso base: B[0,w] = 0, B[k,0] = 0

Podemos usar una tabla de tamaño (n+1)x(W+1) para almacenar los valores B[k,w]

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

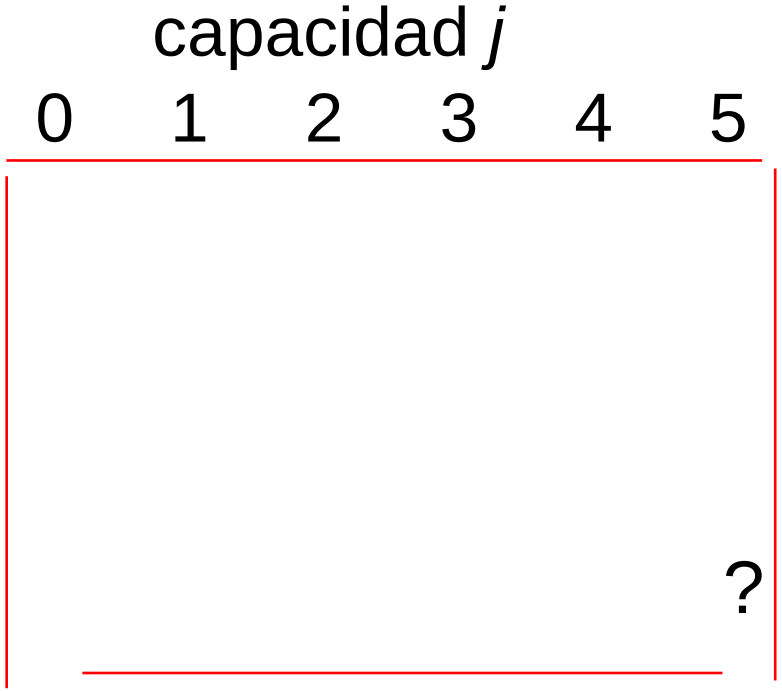
item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$



Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12			
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12		
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10				
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12			
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22		
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10				
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12			
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22		
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10				?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15			?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25		?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	?

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	capacidad j					
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Algoritmo *01Knapsack*(*S, W*):

Input: Conjunto ***S de n*** objetos, beneficio ***bi*** y peso ***wi***; capacidad máxima ***W***

```
for w ← 0 to W do
    B[0,w] = 0
for k ← 1 to n do
    B[k,0] = 0
    for w ← 1 to wk -1 do
            B[k,w] = B[k-1,w]
   for w ← wk to W do
            if B[k-1,w −wk] + bk > B[k-1,w] then
                    B[k,w] = B[k-1,w −wk] + bk
            else B[k,w] = B[k-1,w]
return B[n,W]
```

Búsqueda de la Solución Optimal

- A partir de la matriz calculada por el algoritmo anterior

Si *B*[k,*w*] == *B[k-1,w]*

entonces el objeto k no se selecciona y se pasa a estudiar el objeto k-1 para una mochila de capacidad w:

Problema B[k-1,w]

Si *B*[k,*w*] != *B[k-1,w]*

se selecciona el objeto k, pasando a estudiar el objeto k-1 con una mochila de capacidad w-wk:

Problema B[k-1,w-wk]

Problema Mochila: Ejemplo

Mochila de capacidad $W = 5$

item	peso	ganancia
1	2	$12
2	1	$10
3	3	$20
4	2	$15

$w_1 = 2,\ v_1 = 12$

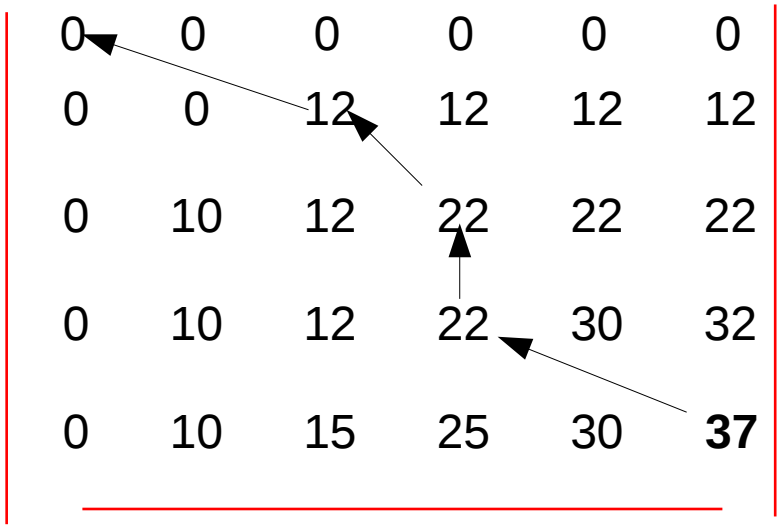
$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacidad j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**



Se seleccionan los objetos 4, 2 y 1