Tecnológico de Monterrey.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

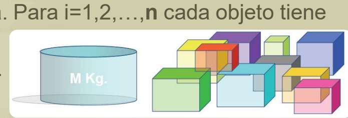
63

Grafos

Algoritmo de la mochila

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



- **Objetivo:** llenar la mochila, maximizando el beneficio obtenido por los objetos transportados y respetando la limitación de la capacidad M.
- **Consideración:** los objetos se pueden fraccionar en trozos (*de no ser así, el problema se torna mucho más difícil*).

Analizando

- **Datos del problema:**
  - **n:** número de objetos disponibles.
  - **M:** capacidad de la mochila.
  - $\mathbf{p = (p_1, p_2, ..., p_n)}$ pesos de los objetos.
  - $\mathbf{b = (b_1, b_2, ..., b_n)}$ beneficios de los objetos.
- **Representación de la solución:**
  - Una solución será de la forma $S = (x_1, x_2, ..., x_n)$, con $0 \le x_i \le 1$, siendo cada x_i la fracción escogida del objeto i.

Formulación matemática:

- Maximizar $\sum_{i=1}^{n} X_i b_i$

para i=1,2,…***n***

- Sujeto a la restricción

$$\sum_{i=1}^{n} X_i p_i \le M \text{ y } 0 \le X_i \le 1 \text{ para } i = 1..n$$

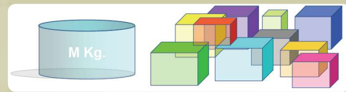
64

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

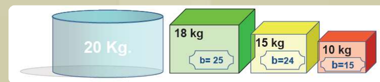
- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



- **Objetivo:** llenar la mochila, maximizando el beneficio obtenido por los objetos transportados y respetando la limitación de la capacidad M.
- **Consideración:** los objetos se pueden fraccionar en trozos (*de no ser así, el problema se torna mucho más difícil*).

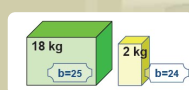
Ejemplo:

Si **n** = 3, **M** = 20, **p** = (18, 15, 10), **b** = (25, 24, 15)



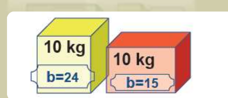
Solución 1:

- S = (1, 2/15, 0)
- Beneficio total = 25 + 24*2/15 = 28,2



Solución 2: 10/15

- S = (0, 2/3, 1)
- Beneficio total = 24*2/3 + 15 = 31

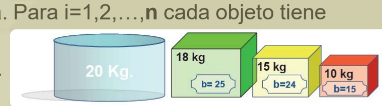


Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



Ejemplo:

Si **n** = 3, **M** = 20, **p** = (18, 15, 10), **b** = (25, 24, 15)

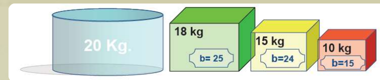
- El problema se ajusta para la aplicación de un algoritmo voraz:
  - **Conjunto de candidatos**: cada uno de los *n* objetos de partida.
  - **Función solución**: tendremos una solución si hemos introducido en la mochila el peso máximo M, o si se han acabado los objetos.
  - **Función seleccionar**: escoger el objeto "más prometedor".
  - **Función de factibilidad**: será siempre cierta (*se pueden añadir trozos*).
  - **Añadir o insertar a la solución**: añadir objeto entero o un trozo.
  - **Función objetivo**: maximizar la suma de los beneficios de cada candidato por la proporción seleccionada del mismo.
- Sólo falta aclarar cuál sería el objeto "más prometedor".

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



Ejemplo:

Si **n** = 3, **M** = 20, **p** = (18, 15, 10), **b** = (25, 24, 15)

Algoritmo

```
public mochila (double M, int[] b, int[] p, int [] x, int n) {
    para i=1...n {
        x[i]← 0 }
    pesoAct ← 0
    // bucle voraz:
    mientras ( pesoAct < M) {
        i← elegir el mejor objeto restante
        si (pesoAct + p[i] ≤ M ) entonces {
            x[i] ← 1
            pesoAct= pesoAct+ p[i]
        } de lo contrario {
            x[i]← (M -pesoAct)/p[i]
            pesoAct= M
        }
    }
}
```

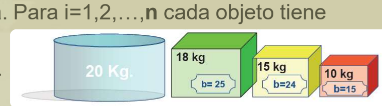

67

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



Ejemplo:

Si **n** = 3, **M** = 20, **p** = (18, 15, 10), **b** = (25, 24, 15)

¿Cuál es el "mejor objeto restante"?

Posibles criterios:

1. El objeto con más beneficio $\mathbf{b_i= Max\{b_1, b_2, \ldots, b_n\}}$ ➡ *(para obtener mayor beneficio)*
2. El objeto menos pesado $\mathbf{p_i=Min\{p_1, p_2, \ldots, p_n\}}$ ➡ *(para poder añadir muchos objetos)*

- ¿Cuál es el mejor criterio de selección?
- ¿Cuál garantiza la solución óptima?
- ¿Se te ocurre algún otro criterio?

Otro criterio de selección:

3. El objeto con mejor proporción $\mathbf{b_i/p_i}$ *(coste por unidad de peso)* $\mathbf{Max\{b_1/p_1, b_2/p_2, \ldots, b_n/p_n\}}$

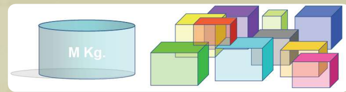
68

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



- **Objetivo:** llenar la mochila, maximizando el beneficio obtenido por los objetos transportados y respetando la limitación de la capacidad M.

Ejemplo:

Si **n** = 5, **M** = 100, **p**=(10,20,30,40,50), **b**=(20,30,66,40,60) y **b/p**=(2.0,1.5,2.2,1.0,1.2)

Analizando

- Aplicando cada uno de los criterios de selección, se obtendrían las siguientes fracciones de los pesos x_i:

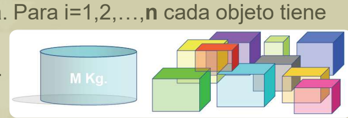
Seleccionar	X_i					Valor
Max b_i	0	0	1	0.5	1	146
Min p_i	1	1	1	1	0	156
Max b_i/p_i	1	1	1	0	0.8	164

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



- **Objetivo:** llenar la mochila, maximizando el beneficio obtenido por los objetos transportados y respetando la limitación de la capacidad M.

Ejemplo:

Si **n** = 5, **M** = 100, **p**=(10,20,30,40,50), **b**=(20,30,66,40,60) y **b/p**=(2.0,1.5,2.2,1.0,1.2)

Complejidad

- Para el algoritmo externo voraz:
  - Tamaño de la entrada: n
  - Operación básica: comparaciones con M ➡ **W(n) = 2n**
- Observa que existe una búsqueda interna (*el arreglo p_i no está ordenado*)
  - Tamaño de la entrada: n
  - Operación básica: Comparación con el máximo. ➡ **W(n)=n**
- Entonces el algoritmo completo tendría una complejidad de:

$$W(n) = 2n + n + n + \ldots + n$$

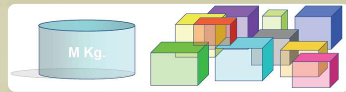
$$W(n) = 2n + \sum_{i=1}^{n} n = 2n + n^2$$

Técnicas de diseño de algoritmos

Algoritmos voraces (*Greedy*)

Ejercicio: Problema de la mochila

- Supón que nos dan ***n*** objetos y una mochila. Para i=1,2,…,**n** cada objeto tiene un peso positivo $\mathbf{p_i}$ y un beneficio $\mathbf{b_i}$.
- La mochila puede llevar un peso máximo **M**.



- **Objetivo:** llenar la mochila, maximizando el beneficio obtenido por los objetos transportados y respetando la limitación de la capacidad M.

Ejemplo:

Si **n** = 5, **M** = 100, **p**=(10,20,30,40,50), **b**=(20,30,66,40,60) y **b/p**=(2.0,1.5,2.2,1.0,1.2)

Complejidad

- Para el algoritmo externo voraz:
  - Tamaño de la entrada: n
  - Operación básica: comparaciones con M → **W(n) = 2n**
- Algoritmo interno de búsqueda (*el arreglo sí está ordenado*)
  - Tamaño de la entrada: **n**
  - Operación básica: Comparación con un elemento del arreglo: → **W(n)=1**
- Entonces el algoritmo completo tendría una complejidad de: $\mathrm{W(n)} = \mathrm{n}\log_2 \mathrm{n} + 2\mathrm{n} + \mathrm{n}$

Alternativa: Ordenar las proporciones una vez y así el buscar no cuesta tanto.

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio: Problema de la mochila

- **Datos del problema:**
  - **n:** número de TIPOS de objetos disponibles
  - **M:** capacidad de la mochila.
  - $\mathbf{u = (u_1, u_2, ..., u_n)}$ unidades de los objetos..
  - $\mathbf{b = (b_1, b_2, ..., b_n)}$ beneficios de los objetos.
- **Representación de la solución:**
  - Una solución será de la forma S = $(x_1, x_2, ..., x_n)$, con $x_i \in$ **u**, siendo cada x_i unidades del objeto i.

Formulación matemática:

- Maximizar $\sum_{i=1}^{n} X_i b_i$

para i=1,2,…***n***

- Sujeto a la restricción

$$\sum_{i=1}^{n} X_i u_i \leq M \quad \text{para } i = 1..n$$

Ejemplo:

Si **n** = 4,

M = 8,

u = (2, 3, 4, 5) unidades de los objetos,

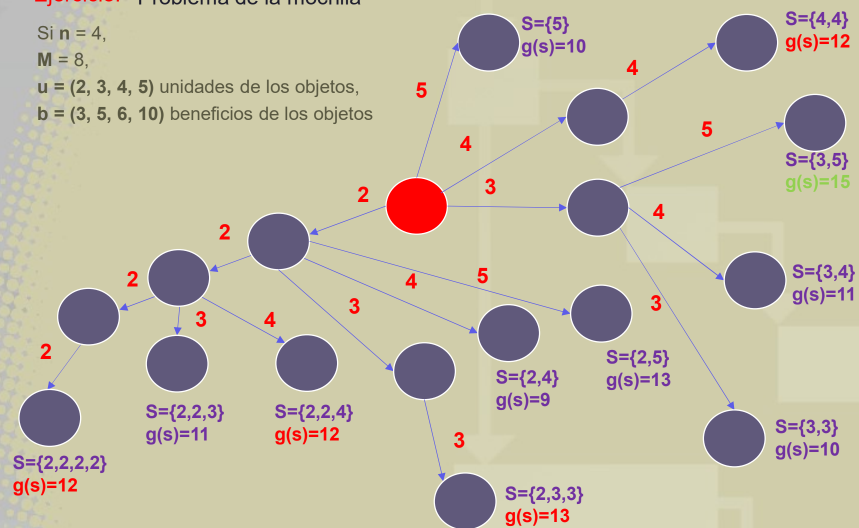
b = (3, 5, 6, 10) beneficios de los objetos

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Ejercicio: Problema de la mochila

Si **n** = 4,
M = 8,
u = (2, 3, 4, 5) unidades de los objetos,
b = (3, 5, 6, 10) beneficios de los objetos



73

Técnicas de diseño de algoritmos

Algoritmos Backtracking

Esquema general (*versión recursiva*)

Ejercicio: Problema de la mochila

función MochilaBackRec (u[], b[], Tipo *i*, Capacidad *r*)

```
{ Calcula el valor de la mejor carga que se puede construir
  empleando los elementos de tipos i a n y cuyo peso no sobrepase r }
a:=0
{ Se prueban por turno las clase de objetos admisibles }
para k:=i hasta n hacer
  si u[k] <= r entonces
      a:=max(a, b[k]+MochilaBackRec(u,b,k,r-u[k]))
devolver a
```

MochilaBackRec (1,M)

74

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) (*Branch and Bound*)

Ejercicio: Problema de la mochila

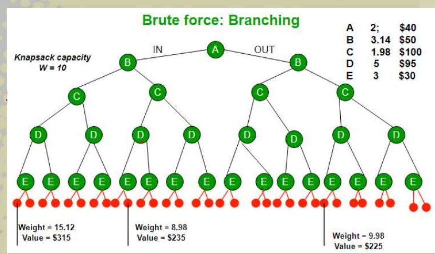
Datos del problema:

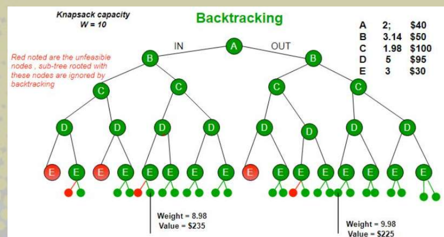
– **n = 5;** número de TIPOS de objetos disponibles

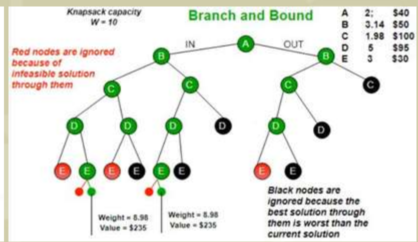
A	2;	$40
B	3.14	$50
C	1.98	$100
D	5	$95
E	3	$30

jetos

objetos







Branch and Bound Algorithm

75

Técnicas de diseño de algoritmos

Algoritmos Ramificación y acotar (poda) (*Branch and Bound*)

Ejercicio: Problema de la mochila

Datos del problema:

– **n = 5;** número de TIPOS de objetos disponibles

– **M = 10;** capacidad de la mochila

– **u = (2, 3.14, 1.98, 5, 3)** unidades de los objetos

– **b = (40, 50, 100, 95, 30)** beneficios de los objetos

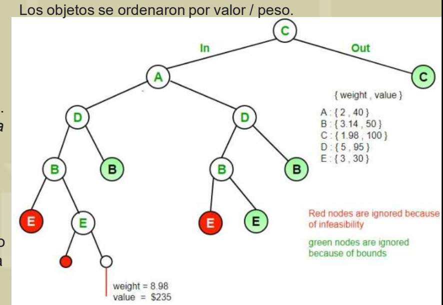
A	2;	$40
B	3.14	$50
C	1.98	$100
D	5	$95
E	3	$30

Solución:

1. Ordena todos los elementos en orden decreciente relacionando el valor por unidad de peso para que se pueda calcular un límite superior utilizando el enfoque codicioso.
2. Inicializa el beneficio máximo, maxProfit = 0
3. Crea una cola vacía, Q.
4. *Crea un nodo ficticio de árbol de decisión y colócalo en Q. La ganancia y el peso del nodo ficticio son 0.*
5. Mientras Q no esté vacío, se realizará:
   - Extraer un elemento de Q, le llamaremos u.
   - Calcula el beneficio del nodo del siguiente nivel. *Si el beneficio es superior a maxProfit, actualiza maxProfit.*
   - Calcula el límite del nodo del siguiente nivel. *Si el límite es mayor que maxProfit, agrega el siguiente nodo de nivel a Q.*
   - Considera que el nodo del siguiente nivel no se considera parte de la solución y agrega un nodo a la cola con el nivel siguiente, pero el peso y la ganancia no considera los nodos del siguiente nivel.

Entrada:
// Para cada par se obtiene el peso de los objetos
// y después el valor del mismo
Elemento arr[ ] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3, 30}};
Capacidad de la mochila W = 10
Salida: El beneficio máximo posible = 235

Los objetos se ordenaron por valor / peso.



Implementation of 0/1 Knapsack using Branch and Bound

76

Grafos

Problema de la mochila *(Knapsack problem)*

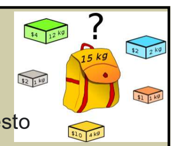
Recordando la programación dinámica (DP)

- DP es un método para resolver cierto tipo de problemas.
- La DP se puede aplicar cuando la solución de un problema incluye soluciones a subproblemas.
- Se necesita encontrar una fórmula recursiva para la solución.
- Se pueden resolver subproblemas de forma recursiva, partiendo del caso trivial, y guardando sus soluciones en memoria.
- Al final, se obtendrá la solución para todo el problema.

Propiedades de un problema que se puede resolver con DP

- Subproblemas simples
  - Se debería poder dividir el problema original en subproblemas más pequeños que tienen la misma estructura.
- Subestructura óptima de los problemas
  - La solución al problema debe ser una composición de soluciones de subproblemas.
- Traslapes
  - Los subproblemas de optimizacion para problemas no relacionados pueden tener subproblemas en común.

Grafos



Problema de la mochila

- Dada una mochila con capacidad máxima **W**, y un conjunto **S** compuesto por ***n*** elementos.
- Cada elemento **i** tiene un peso $\mathbf{w_i}$ y un beneficio $\mathbf{b_i}$ (todos w_i, b_i y W son valores enteros).

Problema:
- ¿Cómo empacar la mochila para lograr maximizar el valor total del beneficio obtenido por los artículos empacados?

Visualizando

Mochila de peso máximo: **W = 20**

W = 20

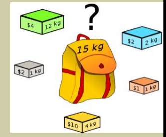
Elementos	Pesos w_i	Beneficios b_i
	2	3
	3	4
	4	5
	5	8
	9	10

Formulación matemática:

$$\max \sum_{i \in T} b_i \quad \text{Sujeto a} \quad \sum_{i \in T} w_i \le W$$

- El problema se llama problema "0-1", porque cada elemento debe ser aceptado o rechazado por completo.
- Existe otra versión de este problema: "Problema de la mochila fraccionada", donde podemos tomar fracciones de los artículos.

Grafos



Problema de la mochila *(Knapsack problem)*

Con el enfoque de fuerza bruta…

Primero solucionemos este problema con un algoritmo sencillo:

- Dado que hay ***n*** elementos, hay 2^n combinaciones posibles de elementos.
- Pasamos por todas las combinaciones y encontramos la que tiene el mayor valor total y con un peso total menor o igual a W.
- El tiempo de ejecución será $O(2^n)$.

¿Podemos resolverlo de mejor forma?

→ Sí, con un algoritmo basado en programación dinámica.

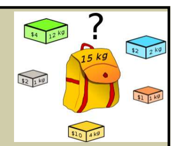
✓ Se necesitan identificar cuidadosamente los subproblemas.

Definiendo los subproblemas…

➢ Si los elementos están etiquetados como *1..n*, entonces un subproblema sería encontrar una solución óptima para S_k = {elementos etiquetados como *1, 2, .. k*}.

- Ésta es una definición de subproblema válida.
- La pregunta es: ¿podemos describir la solución final (S_n) en términos de subproblemas (S_k)?
- Desafortunadamente, no se puede hacer eso.

Grafos



Problema de la mochila *(Knapsack problem)*

Definiendo un subproblema…

➢ Si los elementos están etiquetados como *1..n*, entonces un subproblema sería encontrar una solución óptima para S_k = {elementos etiquetados como *1, 2, .. k*}

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	**?**

Peso máximo: W = 20

Para S_4:
Peso total: 14
Beneficio total: 20

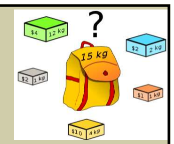
$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=9$ $b_4=10$

Para S_5:
Peso total: 20
Beneficio total: 26

	Elemento	Peso w_i	Beneficio b_i
S_5, S_4	1	2	3
S_5, S_4	2	3	4
S_5, S_4	3	4	5
S_5, S_4	4	5	8
S_5	5	9	10

La solución para S_4 no es parte de la solución para S_5!!!

Grafos



Problema de la mochila *(Knapsack problem)*

Definiendo un subproblema…

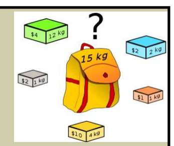
- Entonces, la solución para S_4 no es parte de la solución para S_5.
- Por lo que la definición del subproblema es errónea y se necesita otra.
- Si se agrega otro parámetro: *w*, que representará el peso exacto de cada subconjunto de elementos.
  - El subproblema entonces será calcular *B[k, w]*.

Fórmula recursiva para subproblemas

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Significa que el mejor subconjunto de S_k que tiene un peso total *w* es uno de los siguientes dos:
  1) el mejor subconjunto de S_{k-1} que tiene un peso total *w.*
  2) el mejor subconjunto de S_{k-1} que tiene peso total w-w_k más el elemento *k*.
- El mejor subconjunto de S_k que tiene el peso total *w*, puede contener o no al elemento *k*.
  - Primer caso: $w_k > w$.
    - El elemento *k* no puede ser parte de la solución, ya que si lo fuera, el peso total sería > *w*, lo cual es inaceptable.
  - Segundo caso: $w_k <= w$.
    - El elemento *k* puede estar en la solución, y se elige el caso con mayor valor.

Grafos



Algoritmo de la mochila *(Knapsack problem)*

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        if wi <= w    // el objeto puede ser parte de la solución
            if bi + B[i-1,w-wi] > B[i-1,w]
                B[i,w] = bi + B[i-1,w- wi]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]    // wi > w
```

Tiempo de ejecución

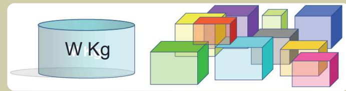
```
for w = 0 to W              O(W)
    B[0,w] = 0
for i = 0 to n              Se repite n veces
    B[i,0] = 0
    for w = 0 to W          O(W)
        < el resto del código >
```

¿Cuál es el tiempo de ejecución de este algoritmo?

O(n*W)

Recuerda que con el algoritmo de *brute-force*, se ejecutaría en $O(2^n)$.

Grafos



Algoritmo de la mochila

Ejemplo:

Si tenemos:
n = 4, número de elementos
W = 5, peso máximo
Elementos (**pesos**, **beneficios**) = (2, 3), (3, 4), (4, 5), (5, 6)

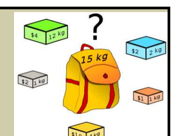
Elementos:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7	7

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
  for w = 0 to W
```

```
if w_i <= w     // el objeto puede ser parte de la solución
   if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_i + B[i-1,w- w_i]
   else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```


Grafos



Problema de la mochila *(Knapsack problem)*

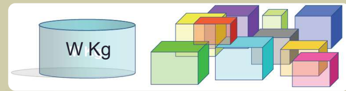
Comentarios

- Este algoritmo solo encuentra el valor máximo posible que se puede llevar en la mochila.
- Para conocer los elementos que hacen este valor máximo, es necesario incorporar algún otro algoritmo.

Conclusiones

- La programación dinámica es una técnica útil para resolver cierto tipo de problemas.
- Cuando la solución se puede describir de forma recursiva en términos de soluciones parciales, se pueden almacenar estas soluciones parciales y reutilizarlas según sea necesario.
- El tiempo de ejecución (algoritmo de programación dinámica *vs* algoritmo *naive*): 0-1 Problema de mochila: O(W*n) *vs* O(2^n).

Grafos



Algoritmo de la mochila

Ejemplo:

Si tenemos:
n = 3, número de elementos
W = 4, peso máximo
Elementos (**pesos**, **beneficios**) = (4, 1), (5, 2), (1, 3)

Elementos:
1: (4,1)
2: (5,2)
3: (1,3)

```
for w = 0 to W
    B[0,w] = 0

for i = 0 to n
    B[i,0] = 0

    for w = 0 to W
      if w_i <= w      // el objeto puede ser parte de la solución
        if b_i + B[i-1,w-w_i] > B[i-1,w]
          B[i,w] = b_i + B[i-1,w- w_i]
        else
          B[i,w] = B[i-1,w]
      else B[i,w] = B[i-1,w]  // w_i > w
```

w \ i	0	1	2	3
0	0	0	0	0
1	0	0	0	3
2	0	0	0	3
3	0	0	0	3
4	0	1	1	3

85