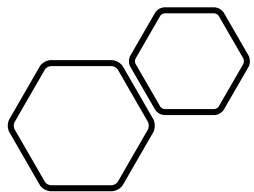


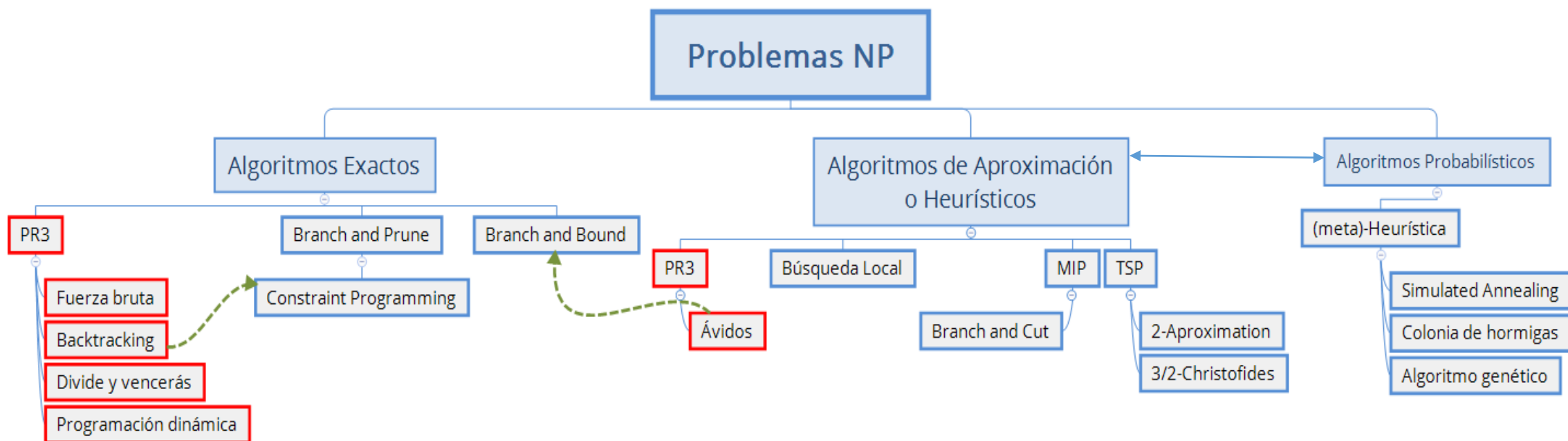


# Algoritmos y Computabilidad

Algoritmos de  
Aproximación 2



# Algoritmos y Computabilidad



$P \neq NP$

- Encontrar soluciones óptimas
  - En tiempo polinómico
  - Para cualquier dimensión del problema
- 
- En general, elegir 2 de las 3

### **Algoritmos de Aproximación**

Intentan acotar la solución a una cierta distancia del óptimo

# Algoritmos de aproximación

En [ciencias de la computación](#) e [investigación de operaciones](#), un **algoritmo de aproximación** es un [algoritmo](#) usado para encontrar soluciones aproximadas a [problemas de optimización](#). Están a menudo asociados con problemas [NP-hard](#); como es poco probable que alguna vez se descubran algoritmos eficientes de [tiempo polinómico](#) que resuelvan exactamente problemas NP-hard, se opta por encontrar soluciones no-óptimas en **tiempo polinomial**. A diferencia de las [heurísticas](#), que usualmente sólo encuentran soluciones razonablemente buenas en tiempos razonablemente rápidos, lo que se busca aquí es encontrar soluciones que está demostrado son de calidad y cuyos tiempos de ejecución están acotadas por **cotas conocidas**... un  **$\rho$ -algoritmo de aproximación** se ha demostrado que la aproximación no será mayor (o menor, dependiendo de la situación) que un factor  $\rho$  veces la solución óptima.

[https://es.wikipedia.org/wiki/Algoritmo\\_de\\_aproximación](https://es.wikipedia.org/wiki/Algoritmo_de_aproximación)

ratio de aproximación o factor de aproximación

# Algoritmos de aproximación

- Esquemas de aproximación en tiempo polinómico (*PTAS Polynomial-Time Approximation Scheme*). Ejemplos:
  - Problema del agente viajero (TSP)
  - Problema de la mochila
  - ...
- Problemas MAX SNP. Para cualquier problema que pertenezca a la clase MAX SNP no existe un PTAS para el problema a menos que  $P=NP$ . Ejemplos:
  - Problema de satisfacibilidad (*maximum satisfiability problem*)
  - Problema de corte máximo (*maximum cut problem*)

Strict NP



[https://en.wikipedia.org/wiki/SNP\\_\(complexity\)#MaxSNP](https://en.wikipedia.org/wiki/SNP_(complexity)#MaxSNP)

# Algoritmos aproximados

- Hemos visto Algoritmos: 2-Aproximado, 3/2-Aproximado para TSP
- Existen también algoritmos (PTAS) que garantizan:
  - $1+\varepsilon$  Aproximado (para el caso de querer minimizar)
  - $1-\varepsilon$  Aproximado (cuando queremos maximizar)
- Esquemas de aproximación en tiempo polinómico (**PTAS Polynomial-Time Approximation Scheme**). Si la dimensión del problema es  $n$ , el algoritmo depende de  $n$  y de  $\varepsilon$  y es polinómico en  $n$
- **FPTAS (Fully PTAS)**. Es PTAS (Depende de  $n$  y de  $\varepsilon$  y es polinómico en  $n$ ) y también es polinómico en  $\varepsilon$ . Muy pocos problemas son FPTAS.

# Algoritmo 1/2-Aproximado mochila 0-1

- Supongamos A1 un algoritmo que coge los elementos de la mochila en orden decreciente de  $\frac{v_i}{w_i}$  (valor/peso)
- Y A2 un algoritmo que selecciona los elementos por orden decreciente del valor,  $v_i$
- Entonces, el algoritmo A3, que coge el máximo(A1,A2)  $\geq OPT/2$ . Esto significa que A3 es 2-aproximado

# Demostración

- Supongamos que con A1 cogemos los elementos  $\frac{v_i}{w_i}$  en orden decreciente hasta que llegamos al ítem x donde la suma de los pesos excede el valor de la mochila. Y añadimos el elemento x a la lista de los elementos escogidos. Obviamente la suma de estos pesos excederá el límite de la mochila W, y la suma del valor de los elementos escogidos  $(v_1 + v_2 + \dots + v_x)$  debe ser  $\geq \text{OPT}$ .



# Demostración

- $(v_1 + v_2 + \dots + v_x)$  debe ser  $\geq \text{OPT}$ .
- Por otro lado,  $v_x$  debe ser  $\leq A_2$ , por cuanto  $A_2$  elige sus elementos por orden decreciente del valor, y si hubiese sido mayor lo habría elegido en primer lugar.

# Demostración

- $(v_1 + v_2 + \dots + v_x)$  debe ser  $\geq OPT$ .
- Por otro lado,  $v_x$  debe ser  $\leq A_2$ , por cuanto  $A_2$  elige sus elementos por orden decreciente del valor, y si hubiese sido mayor lo habría elegido en primer lugar.

- Por tanto:

$$\begin{aligned} & \overbrace{(v_1 + v_2 + \dots + v_x)}^{A_1} \geq OPT \\ & \Rightarrow (A_1 + v_x) \geq OPT \\ & \Rightarrow (A_1 + A_2) \geq OPT \text{ (Puesto que } A_2 \geq v_x) \end{aligned}$$

# Demostración

- Por tanto:

$$\begin{aligned}(v_1 + v_2 + \dots + v_x) &\geq OPT \\ \Rightarrow (A_1 + v_x) &\geq OPT \\ \Rightarrow (A_1 + A_2) &\geq OPT \text{ (Puesto que } A_2 \geq v_x)\end{aligned}$$

- Esto significa que  $A_1$  o  $A_2$  es  $\geq \frac{OPT}{2}$ , por cuanto la suma es  $\geq OPT$

# Demostración

- Esto significa que A1 o A2 es  $\geq \frac{OPT}{2}$ , por cuanto la suma es  $\geq OPT$
- Como el algoritmo A3 elige el máximo de A1 y A2:  
A3 será  $\geq \frac{OPT}{2}$  c.q.d.

A3 es un algoritmo 1/2-Approximado

# Problema en el aula

- En el problema de la mochila 1-0:

¿Es posible que un algoritmo avaricioso encuentre siempre una aproximación superior a un porcentaje  $\rho$  de la solución óptima, si cada ítem pesa como mucho un 2% de la capacidad de la mochila?

En caso de que la respuesta sea afirmativa,  
¿Cuánto podría valer  $\rho$ ?



# Definición básica (para problemas de minimización)

- Un algoritmo es  $\rho$  -Aproximado, para algún  $\rho > 1$
- Si  $Alg(I) \leq \rho \cdot OPT(I)$ , para todas las entradas  $I$
- Algunos problemas pueden obtener soluciones con  $\rho$  tan próximo a 1 como se quiera. Es decir, que podemos aproximar al óptimo cuanto queramos.



# PTAS (Polynomial-Time Approximation Schemes)

- Los algoritmos PTAS tienen 2 parámetros:
  - Datos de entrada,  $I$
  - Parámetro  $\varepsilon > 0$
- $Alg(I, \varepsilon) \leq (1 + \varepsilon) \cdot OPT(I)$
- El tiempo de ejecución es polinómico para cualquier instancia de datos,  $I$

Problemas de minimización

- Ejemplos válidos:

$$O\left(\left(\frac{1}{\varepsilon}\right)^4 n^2\right), \quad O\left(2^{\frac{1}{\varepsilon}} \cdot n \cdot \log n\right), \quad O(n^{2/\varepsilon})$$

- Ejemplo no válido

$$O((1/\varepsilon) \cdot 2^n)$$

# FPTAS (Fully Polynomial-Time Approximation Schemes)

- $Alg(I, \varepsilon) \leq (1 + \varepsilon) \cdot OPT(I)$
- El tiempo de ejecución es polinómico para cualquier instancia de datos,  $I$
- Ejemplos válidos:

$$O\left(\left(\frac{1}{\varepsilon}\right)^4 n^2\right),$$

Polinómico en  $\frac{1}{\varepsilon}$



PTAS, FPTAS

$$O\left(2^{\frac{1}{\varepsilon}} \cdot n \cdot \log n\right), \quad O(n^{2/\varepsilon})$$


No polinómico en  $\frac{1}{\varepsilon}$



PTAS



# PTAS (Polynomial-Time Approximation Schemes)

- Los algoritmos PTAS tienen 2 parámetros:
    - Datos de entrada,  $I$
    - Parámetro  $\varepsilon > 0$
  - $Alg(I, \varepsilon) \leq (1 - \varepsilon) \cdot OPT(I)$
  - El tiempo de ejecución es polinómico para cualquier instancia de datos,  $I$
- Problemas de maximización
- 

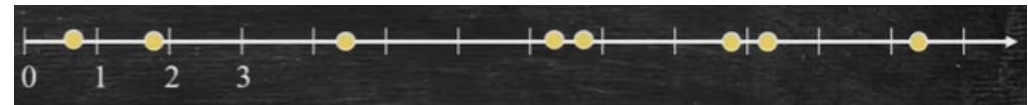
# PTAS Problema de la Mochila

- Problema de la mochila
  - $X = \{x_1, x_2, \dots, x_n\}$ 
    - Pesos:  $weight(x_i)$
    - Valor:  $V, value(x_i)$
  - $W$ , capacidad de la mochila

## Programación dinámica

- Coste computacional:  $O(n \cdot V_{Total})$
- Funciona con valores enteros

¿Qué podemos hacer si los valores son reales?



# Programación dinámica

*IntegerValueKnapsack*( $X, W$ )

```
1: Let  $A[0..n, 0..V_{\text{tot}}]$  be an array, where  $V_{\text{tot}} = \sum_{i=1}^n \text{value}(x_i)$ .
2: for  $i \leftarrow 0$  to  $n$  do
3:    $A[i, 0] \leftarrow 0$ 
4: end for
5: for  $j \leftarrow 1$  to  $V_{\text{tot}}$  do
6:    $A[0, j] \leftarrow \infty$ 
7: end for
8: for  $i \leftarrow 1$  to  $n$  do
9:   for  $j \leftarrow 1$  to  $V_{\text{tot}}$  do
10:    if  $\text{value}(x_i) \leq j$  then
11:       $A[i, j] \leftarrow \min(A[i-1, j], \text{weight}(x_i) + A[i-1, j - \text{value}(x_i)])$ 
12:    else
13:       $A[i, j] \leftarrow A[i-1, j]$ 
14:    end if
15:  end for
16: end for
17:  $\text{OPT} \leftarrow \max\{j : 0 \leq j \leq V_{\text{tot}} \text{ and } A[n, j] \leq W\}$ 
18: Using  $A$ , find a subset  $S \subseteq X$  of value  $\text{OPT}$  and total weight at most  $W$ .
19: return  $S$ 
```

*ReportSolution*( $X, A, \text{OPT}$ )

```
1:  $j \leftarrow \text{OPT}$ ;  $S \leftarrow \emptyset$ 
2: for  $i \leftarrow n$  downto 1 do
3:   if  $\text{value}(x_i) \leq j$  then
4:     if  $\text{weight}(x_i) + A[i-1, j - \text{value}(x_i)] < A[i-1, j]$  then
5:        $S \leftarrow S \cup \{x_i\}$ ;  $j \leftarrow j - \text{value}(x_i)$ 
6:     end if
7:   end if
8: end for
9: return  $S$ 
```

## Recurrencia

$$\begin{cases} 0 & \text{si } v = 0 \\ \infty & \text{si } i = 0 \text{ y } v > 0 \\ \text{opt}(i - 1, v) & \text{si } v_i > v \\ \min\{\text{opt}(i - 1, v), w_i + \text{opt}(i - 1, v - v_i)\} & \text{en otro caso} \end{cases}$$

Esta función determina **el peso mínimo necesario** para obtener **exactamente un valor  $v$**  usando un subconjunto de los primeros  $i$  ítems.

(En lugar de calcular el valor máximo con un peso dado, esta formulación se centra en minimizar el peso para obtener un valor dado)

$$\begin{cases} 0 & \text{si } v = 0 \\ \infty & \text{si } i = 0 \text{ y } v > 0 \\ \text{opt}(i - 1, v) & \text{si } v_i > v \\ \min\{\text{opt}(i - 1, v), w_i + \text{opt}(i - 1, v - v_i)\} & \text{en otro caso} \end{cases}$$

1. **Caso base 1:**  $\text{opt}(i, v) = 0$  si  $v = 0$

Esto significa que si no necesitamos obtener ningún valor (es decir,  $v = 0$ ), el peso mínimo necesario es 0, independientemente de cuántos ítems tengamos disponibles.

2. **Caso base 2:**  $\text{opt}(i, v) = \infty$  si  $i = 0$  y  $v > 0$

Si no tenemos ítems disponibles (es decir,  $i = 0$ ) pero necesitamos obtener un valor positivo  $v$ , esto es imposible, y por lo tanto, representamos esto como un peso infinito.

$$\begin{cases} 0 & \text{si } v = 0 \\ \infty & \text{si } i = 0 \text{ y } v > 0 \\ \text{opt}(i - 1, v) & \text{si } v_i > v \\ \min\{\text{opt}(i - 1, v), w_i + \text{opt}(i - 1, v - v_i)\} & \text{en otro caso} \end{cases}$$

3. **Caso recursivo 1:**  $\text{opt}(i, v) = \text{opt}(i - 1, v)$  si  $v_i > v$

Si el valor del ítem actual  $v_i$  es mayor que el valor objetivo  $v$ , no podemos incluir este ítem en el subconjunto porque superaría el valor que estamos tratando de alcanzar.

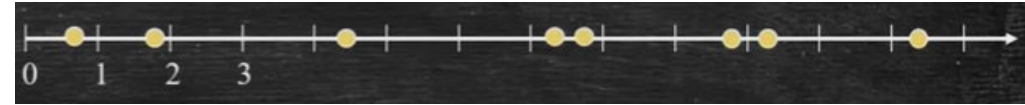
Por lo tanto, simplemente miramos la solución para los primeros  $i - 1$  ítems con el mismo valor objetivo  $v$ .

$$\begin{cases} 0 & \text{si } v = 0 \\ \infty & \text{si } i = 0 \text{ y } v > 0 \\ \text{opt}(i - 1, v) & \text{si } v_i > v \\ \min\{\text{opt}(i - 1, v), w_i + \text{opt}(i - 1, v - v_i)\} & \text{en otro caso} \end{cases}$$

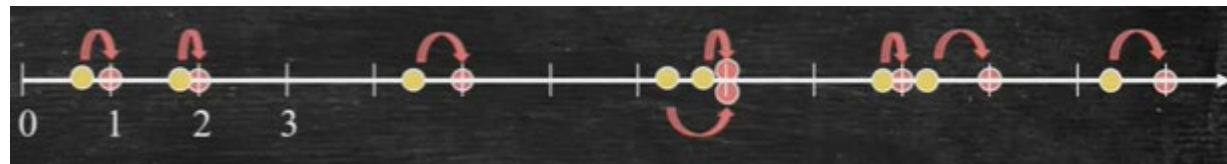
4. **Caso recursivo 2:**  $\text{opt}(i, v) = \min\{\text{opt}(i - 1, v), w_i + \text{opt}(i - 1, v - v_i)\}$  de otra manera

En el caso general, tenemos dos opciones: no incluir el ítem  $i$  actual, o incluirlo. Si no lo incluimos, simplemente buscamos la solución para los primeros  $i - 1$  ítems (esto es  $\text{opt}(i - 1, v)$ ). Si decidimos incluir el ítem  $i$ , debemos agregar su peso  $w_i$  al peso total y buscar la solución para los primeros  $i - 1$  ítems con el valor restante  $v - v_i$  (esto es  $w_i + \text{opt}(i - 1, v - v_i)$ ). Tomamos el mínimo de estas dos opciones para encontrar el peso óptimo que cumple el valor objetivo.

# Estrategia 1



- Reemplazar todos los valores de los ítems, por sus redondeos superiores



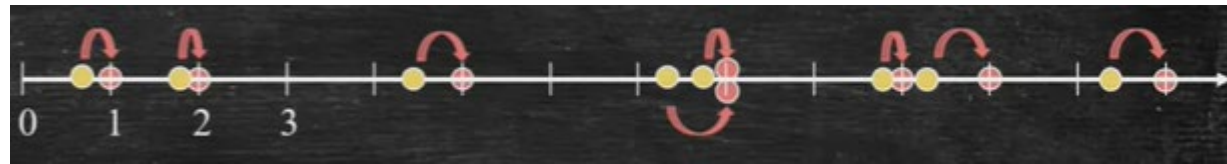
$$value^*(x_i) = \lceil value(x_i) \rceil$$

- Resolver mediante programación dinámica con los nuevos valores
- El resultado obtenido de esta forma puede ser una buena aproximación al óptimo (no hemos modificado los pesos)



# Problemas de la Estrategia 1

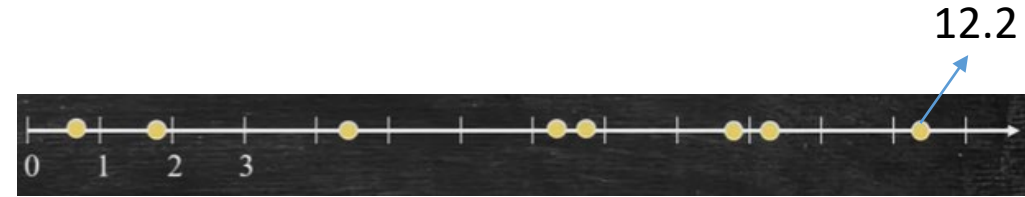
- Reemplazar todos los valores de los ítems, por sus redondeos superiores



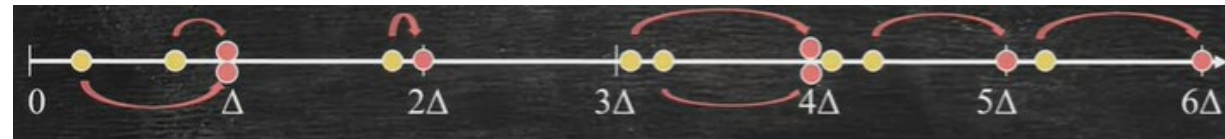
$$value^*(x_i) = \lceil value(x_i) \rceil$$

- La suma de los valores  $V_{Total}^*$  puede ser muy grande
- No tenemos control sobre el ratio de aproximación

# Estrategia 2

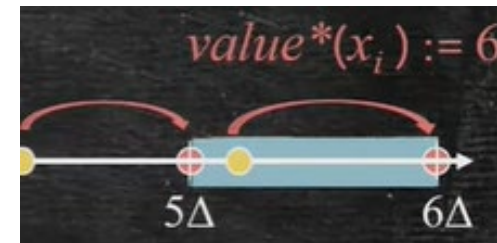


- Igual a la Estrategia 1, salvo el primer paso
- Reemplazar todos los valores de los ítems por un múltiplo de  $\Delta$



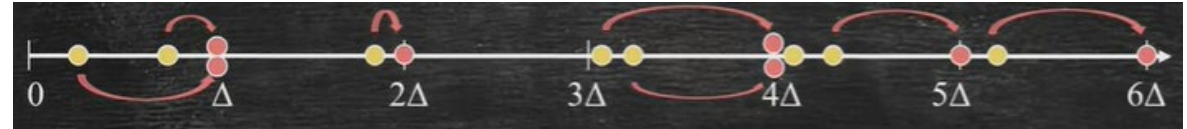
- Y escalamos los valores dividiendo por  $\Delta$  (con el fin de obtener valores de ítem más pequeños)

$$value^*(x_i) = \left\lfloor \frac{value(x_i)}{\Delta} \right\rfloor$$

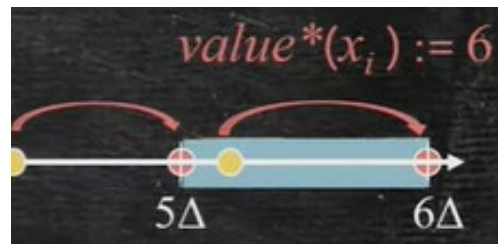


# ¿Cómo podemos obtener el valor de $\Delta$ ?

- $value^*(x_i) = \left\lceil \frac{value(x_i)}{\Delta} \right\rceil$



- $PTAS \geq (1 - \varepsilon) \cdot OPT = OPT - \underbrace{\varepsilon \cdot OPT}_{\text{Máximo error permitido}}$



$\Delta$  Máximo error introducido por cada ítem

# ¿Cómo podemos obtener el valor de $\Delta$ ?

Estamos buscando:

$$\text{Error total} \leq \varepsilon \cdot OPT$$

Y sabemos:

$$\text{Error de cada ítem} \leq \Delta$$

En cada solución hay como mucho  $n$  ítems

$$\left. \begin{array}{l} \text{Error de cada ítem} \leq \Delta \\ \text{En cada solución hay como mucho } n \text{ ítems} \end{array} \right\} \text{Error total} \leq n \cdot \Delta$$

Para obtener  $\Delta$ :

$$n \cdot \Delta = \varepsilon \cdot OPT \Rightarrow \Delta = (\varepsilon/n) \cdot OPT$$

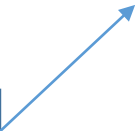
# Estrategia 2

- Elegimos  $\Delta = (\varepsilon/n) \cdot OPT$
- Reemplazar todos los valores de los ítems, por sus redondeos superiores, divididos por delta

$$value^*(x_i) = \left\lceil \frac{value(x_i)}{\Delta} \right\rceil$$

- Resolver mediante programación dinámica con los nuevos valores
- El resultado obtenido de esta forma puede ser una buena aproximación al óptimo (no hemos modificado los pesos)


# Estrategia 2

- Elegimos  $\Delta = (\varepsilon/n) \cdot \boxed{OPT}$   No sabemos cuanto vale OPT
- Reemplazar todos los valores de los ítems, por sus redondeos superiores

$$value^*(x_i) = \left\lceil \frac{value(x_i)}{\Delta} \right\rceil$$

- Resolver mediante programación dinámica con los nuevos valores
- El resultado obtenido de esta forma puede ser una buena aproximación al óptimo (no hemos modificado los pesos)

# Estrategia 2

- Elegimos  $\Delta = (\varepsilon/n) \cdot \boxed{OPT}$   No sabemos cuanto vale OPT
- Reemplazamos el OPT por algún valor que como mínimo tenga que valer la mochila, LB (*Lower Bound*)
- Si suponemos que todos los pesos de los ítems son menores que la capacidad de la mochila (si no, los eliminamos), podemos coger como LB el ítem de la mochila con mayor valor

$$LB = \max_{x_i \in X} value(x_i)$$

# Algoritmo Final

- Elegimos  $\Delta = (\varepsilon/n) \cdot LB$ , donde  $LB = \max_{x_i \in X} value(x_i)$
- Reemplazar todos los valores de los ítems, por sus redondeos superiores

$$value^*(x_i) = \left\lceil \frac{value(x_i)}{\Delta} \right\rceil$$

- Resolver mediante programación dinámica con  $value^*$
- Computar el subconjunto de ítems obtenidos con los valores originales





```

5 @author: jqunteiro
6 """
7 import math
8
9 value = [12, 30, 7, 8,20]
10 weight = [2, 4, 3, 5,3]
11 W = 10 #
12 epsilon = 0.5
13
14 n = len(value) # n = 5
15
16 delta = (epsilon/n)*max(value) # delta = 3
17
18
19 new_value = list(map(lambda x: math.ceil(x/delta), value))
20
21 print(new_value) # new_value = [4,10,3,3,7]

```

```

3 int: n=5;
4 set of int: ITEMS = 1..n;
5
6
7 array[ITEMS] of int: value= [12, 30, 7, 8,20];
8 %array[ITEMS] of int: value= [4,10,3,3,7];
9
10 array[ITEMS] of int: weight = [2, 4, 3, 5,3];
11 int: w=10;
12
13
14 array[ITEMS] of var 0..1: x;
15 var int: obj = sum(i in ITEMS)(value[i] * x[i]);
16
17 constraint sum(i in ITEMS)(weight[i] * x[i]) <= w;
18 solve maximize obj;
19
20 output ["x = ", show(x), " obj=", show(obj),"\n"];

```

[12, 30, 7, 8,20]	x = [1, 1, 0, 0, 1] obj=62
	-----
	=====

[4,10,3,3,7]	x = [1, 1, 0, 0, 1] obj=21
	-----
	=====