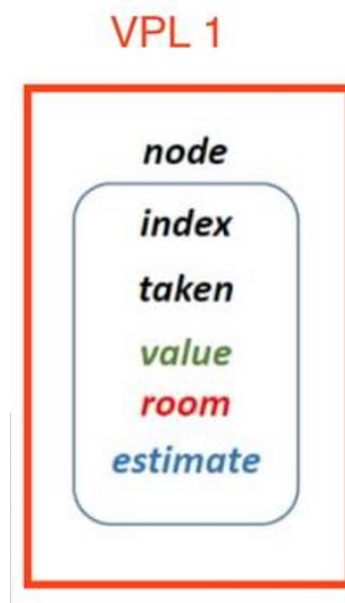


# Algoritmos y Programación

Práctica 5.1: Branch &  
Bound (Primera Parte)

## Ejercicios (1/2)

- Dos ejercicios:
  - 1) Declaración de la clase 'node' que vamos a utilizar para programar nuestro recorrido en profundidad.



# Formato del fichero de entrada

- La primera línea es un descriptor: número de items, peso máximo
- El resto de las líneas tiene el valor y el peso de cada item

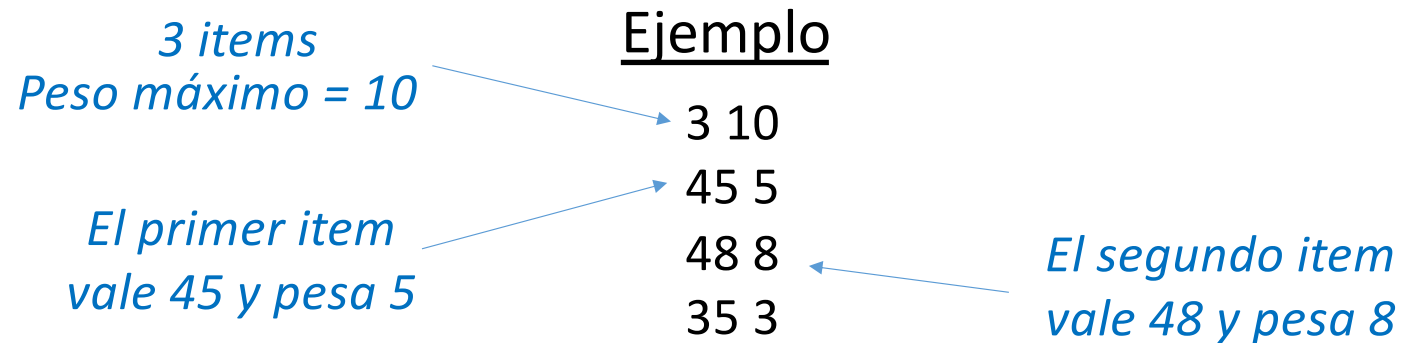
*3 items*  
*Peso máximo = 10*

Ejemplo

*El primer item*  
*vale 45 y pesa 5*

*El segundo item*  
*vale 48 y pesa 8*

3 10  
45 5  
48 8  
35 3



# VPL 1

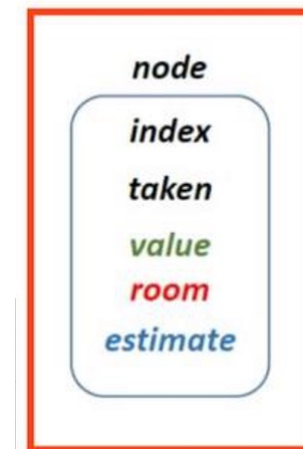
## main.py

```
1 from collections import namedtuple
2 from node import *
3
4 first_line = input().split()
5 item_count = int(first_line[0])
6 capacity = int(first_line[1])
7
8 items = []
9 for i in range(1, item_count+1):
10     line = input()
11     parts = line.split()
12     items.append(Item(i-1, int(parts[0]), int(parts[1])))
13
14 node1 = Node(1, [], 0, capacity)
15 node2 = Node(2, [1], items[1].value, capacity - items[1].weight)
16 node3 = Node(3, [2], items[2].value, capacity - items[2].weight)
17
18 print("Node 1: " + str(node1.index) + " " +
19       str(node1.room) + " " + str(node1.taken) + " " +
20       str(node1.value) + " " + str(node1.estimate(items)))
21 print("Node 2: " + str(node2.index) + " " +
22       str(node2.room) + " " + str(node2.taken) + " " +
23       str(node2.value) + " " + str(node2.estimate(items)))
24 print("Node 3: " + str(node3.index) + " " +
25       str(node3.room) + " " + str(node3.taken) + " " +
26       str(node3.value) + " " + str(node3.estimate(items)))
```

## node.py

```
1 from collections import namedtuple
2
3 Item = namedtuple("Item", ['index', 'value', 'weight'])
4
5 class Node:
6     def __init__(self, index, taken, value, room):
7         return
8
9     def estimate(self, items):
10         return 0
11
```

## VPL 1



<sup>4</sup>  
*Este ejercicio no puntúa*

## Ejercicios (2/2)

### 2) Utilizando nuestra clase 'node' programar el recorrido en profundidad

VPL 2

#### Branch and Bound, DFS

- Empezamos con el nodo raíz y lo metemos en una lista (*append*) de nodos vivos
- Mientras haya nodos en la lista de nodos vivos, hacemos un pop de la lista

• Comprobamos si en el nodo actual hay espacio libre en la mochila

• Comprobamos si la mejor estimación (*bound*) podría mejorar la solución incumbente (el mejor valor obtenido hasta el momento)

• Si el valor obtenido en el nodo actual mejora la mejor solución hasta el momento, **actualizamos el valor de la mejor solución, y los items elegidos para obtener esa solución (*taken*)**

- **Si no hemos llegado al final del árbol:**

- Ramificamos (*branch*) por la derecha (*append*)
- Ramificamos (*branch*) por la izquierda (*append*)

De esta forma, cuando hagamos el *pop* empezaremos a ramificar por la izquierda

VPL 1

*node*  
*index*  
*taken*  
*value*  
*room*  
*estimate*

# VPL 2

## node.py

```
1 # Copia aquí la definición del nodo que resuelve
2 # el VPL anterior!
3
4
```

## main.py

```
1 from collections import namedtuple
2
3 from node import *
4 from solve import *
5
6 first_line = input().split()
7 item_count = int(first_line[0])
8 capacity = int(first_line[1])
9
10 items = []
11 for i in range(1, item_count+1):
12     line = input()
13     parts = line.split()
14     items.append(Item(i-1, int(parts[0]), int(parts[1])))
15
16 _, _, visiting_order = solve_branch_and_bound_DFS(capacity, items, True)
17 print(visiting_order)
```

# VPL 2

solve.py

```
1 from node import *
2
3 def solve_branch_and_bound_DFS(capacity, items, record_visiting_order = False):
4     """
5     :param capacity: capacidad de la mochila
6     :param items: items de la mochila
7     :param record_visiting_order: activa/desactiva el registro de nodos visitados
8     :return: Por ahora sólo devuelve la lista de nodos visitados
9     """
10
11     # Completa este código para realizar el recorrido DFS; tienes
12     # indicados los sitios que debes completar con tres puntos
13     # suspensivos ("...")
14
15     # Utilizamos la lista 'alive' como nuestra pila de nodos vivos
16     # (pendientes de visitar) para programar nuestro recorrido DFS.
17
18     alive = []
19
20     # Utilizamos la lista Visiting_Order como el registro de nodos
21     # visitados (el contenido final de esta lista lo utiliza el VPL
22     # para comprobar que nuestro recorrido DFS es correcto).
23
24     visiting_order = []
```

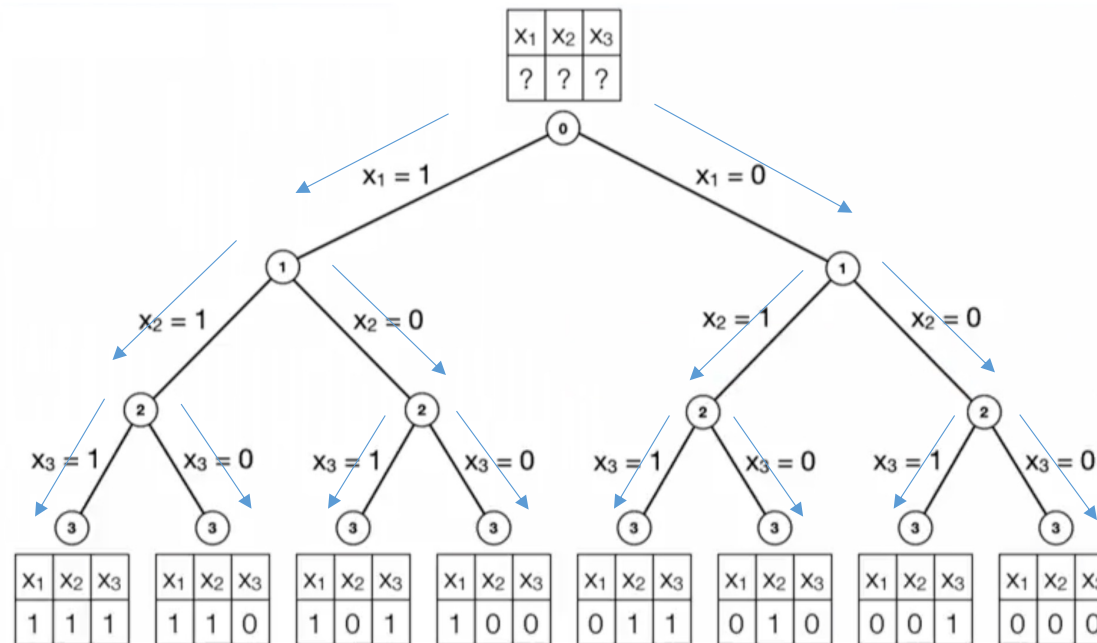
# VPL 2

```
25
26 # 1) Creamos el nodo raiz (en este VPL todavía no utilizamos los
27 #   parámetros taken, value, room, con lo que se inicializan con
28 #   lista vacía y 0). El único valor necesario en el nodo es el
29 #   índice al primer elemento de la lista (index = 0).
30 # ...
31
32 # Lo añadimos a la lista de nodos vivos (alive)
33 # ...
34
35 # Mientras haya nodos en la lista de nodos vivos
36 # ...
37 while True:
38     # Avanzamos al siguiente nodo de nuestro recorrido DFS (hacemos un pop
39     # de la lista) y lo registramos en nuestro recorrido DFS.
40
41     current = alive.pop()
42     if record_visiting_order:
43         visiting_order.append(current.index)
44
45     # Si no hemos llegado al final del árbol
46     #   1) Ramificamos (branch) por la derecha (append)
47     #   2) Ramificamos (branch) por la izquierda (append)
48     # ...
49
50 return 0, [], visiting_order
```



# Formato de la salida del programa

- Muestra el índice de los items visitados en el recorrido en profundidad
- Por ejemplo, éste es el resultado al realizar el recorrido en profundidad completo de 3 items



Salida =[0, 1, 2, 3, 3, 2, 3, 3, 1, 2, 3, 3, 2, 3, 3]