



# Algoritmos y Programación

Práctica 9b: Problema del  
ladrón (*con Programación  
Dinámica*)

# Ejercicio

Utilizando programación dinámica programar mediante memoization y tabulation la recurrencia explicada en clase para resolver el problema del ladrón



Recurrencia:

$$t(n) = \max(t(n-2) + v[n], t(n-1))$$

$$t(n) = 0 \quad : \text{ si } n \leq 0$$

# Ejemplo Fibonacci

$$t_n = f(t_{n-1}, t_{n-2}) = t_{n-1} + t_{n-2} \quad n \geq 2$$
$$t_0 = 0 \quad t_1 = 1$$

```
def Fib(n) {  
    if (n < 2)  
        return n  
    else  
        return Fib(n-2) + Fib(n-1)  
}
```

*Versión Recursiva*

*Versión implementada en  
Python con MEMOIZATION*

```
def Fib(n):  
    mem = {} # Diccionario  
  
    def memFib(n):  
        key = n  
        if key not in mem:  
            if n < 2:  
                r = n  
            else:  
                r = memFib(n-1) + memFib(n-2)  
  
            mem[key] = r  
        return mem[key]  
  
    return memFib(n)
```

## Ejemplo Fibonacci

$$t_n = f(t_{n-1}, t_{n-2}) = t_{n-1} + t_{n-2} \quad n \geq 2$$
$$t_0 = 0 \quad t_1 = 1$$

```
def Fib(n) {  
    if (n < 2)  
        return n  
    else  
        return Fib(n-2) + Fib(n-1)  
}
```

Versión Recursiva

```
def Fib(n):  
    if n < 2:  
        return n  
    else:  
        table = [ ]           # Lista o array  
  
        table.append(0)  
        table.append(1)  
  
        for j in range(2, n+1):  
            table.append(table[j-2] + table[j-1])  
        return table[n]
```

Versión implementada en  
Python con TABULATION

# Formato del fichero de entrada

- La primera línea indica el número de casas
- El resto de las líneas tiene el valor de cada casa

*5 casas* → Ejemplo

5

3

*Robando en la primera casa el ladrón tiene un beneficio de 3*

10

*Robando en la segunda casa el ladrón tiene un beneficio de 10*

3

1

2



# VPL

## main.py

```
1 from solve_memoization import *
2 from solve_tabulation import *
3
4 first_line = input().split()
5 item_count = int(first_line[0])
6
7 items = []
8 for i in range(1, item_count+1):
9     parts = input().split()
10    items.append(int(parts[0]))
11
12 # Comenzamos programando la recurrencia mediante tabulation
13 value1, taken1 = solve_tabulation(items)
14 print(value1)
15 print(taken1)
16
17 # Cuando termines tabulation, comenta el código anterior
18 # para desactivarlo (la llamada a solve_tabulation y los
19 # dos print) y descomenta las siguientes líneas para que
20 # programes la recurrencia mediante memoization.
21 #
22 # value2, taken2 = solve_memoization(items)
23 # print(value2)
24 # print(taken2)
25
26 # Cuando termines los dos ejercicios puedes activar estas
27 # líneas para comprobar que los dos dan exactamente los
28 # mismos resultados.
29 #
30 # assert value1 == value2
31 # assert taken1 == taken2
```

# VPL

## solve\_tabulation.py

```
1 # Recurrencia del problema del ladrón
2 # -----
3 #   t(n) = max (t(n-2) + v[n], t(n-1))
4 #   t(n) = 0                                     : si n<0
5
6 def solve_tabulation(items):
7     table = []
8     taken = []
9
10    def fill_table():
11        # Primera fase: Rellenamos la lista 'table' con las
12        # soluciones de todos los subproblemas (o sea, los
13        # beneficios que puede conseguir el ladrón).
14        # ...
15        return
16
17    def fill_taken():
18        # Segunda fase: Rellenamos la lista 'taken' con el
19        # índice de las casas elegidas por el ladrón para
20        # obtener el máximo beneficio. En el ejemplo de las
21        # transparencias el contenido de esta lista es: [2,5]
22        # (la segunda casa y la quinta casa).
23        # ...
24        return
25
26    fill_table()
27    fill_taken()
28    return 0, taken
```

# VPL

## solve\_memoization.py

```
1 # Recurrencia del problema del ladrón
2 # -----
3 #   t(n) = max (t(n-2) + v[n], t(n-1))
4 #   t(n) = 0                                     : si n<0
5
6 def solve_memoization(items):
7     mem = {}
8     taken = []
9
10    def t(n):
11        # Primera fase: Calculamos la recurrencia guardando en
12        # el diccionario la solución optima de cada subproblema.
13        # ...
14        # Aviso: Para resolver este ejercicio no es valido
15        # utilizar el soporte de @functools
16        return 0
17
18    def fill_taken():
19        # Segunda fase: Rellenamos la lista 'taken' con el
20        # indice de las casas elegidas por el ladrón para
21        # obtener el máximo beneficio. En el ejemplo de las
22        # transparencias el contenido de esta lista es: [2,5]
23        # (la segunda casa y la quinta casa).
24        # ...
25        return
26
27    n = len(items) - 1
28
29    max_benefit = t(n)
30    fill_taken()
31
32    return 0, taken
```