Programação Dinâmica – II
Problema da mochila ("knapsack problem")

- Conjunto A de n objectos, $A = \{1, 2, \ldots, n\}$, cada tem um "valor" v_i e um "tamanho" t_i.
- Mochila com um tamanho T.
- Problema: determinar um conjunto $B \subseteq A$ de objectos que cabe na mochila e maximiza o valor "transportado".

$$\boxed{\sum_{i \in B} v_i \text{ máximo}} \text{ com a restrição } \boxed{\sum_{i \in B} t_i \leq T}$$

Por exemplo

A	problemas propostos ao aluno num exame
t_i	tempo que o aluno demora a fazer o problema i
v_i	cotação do problema i
T	tempo total do exame
Problema: $B \subseteq A$	problemas que o aluno deverá resolver, $B \subseteq A$

(supõe-se que cada resposta é valorizada com 100% ou com 0%)
Exemplo. O tempo total do exame é 16 horas (!) (corresponde à capacidade da mochila) e temos

Problema:	1	2	3	4	5	6
Cotação:	8	9	12	15	5	11
Tempo:	2h	5h	6h	8h	3h	5h

Para maximizar a classificação obtida, que problemas deve o aluno resolver?
→ Resposta: os problemas 1, 4 e 6 (valor total 34 em 60).
A função seguinte, escrita em `python`, retorna o valor óptimo e a correspondente lista de elementos.

```
Parâmetros: t:  lista dos tamanhos
            v:  lista dos valores
            i:  próximo elemento a ser ou não seleccionado
            tm: espaço ainda disponível na mochila
Retorna:    (valor,lista óptima)
Chamada:    valor([0,2,5,6,8,3,5],[0,8,9,12,15,5,11],1,15)
            -> (34, [1, 4, 6])    (custo óptimo, solução óptima)
Comentário: o índice 0 das listas não é usado

def valor(t,v,i,tm):
    if i==len(t) or tm<=0:
        return (0,[])
    if t[i]>tm:
        return valor(t,v,i+1,tm) # o elemento i não pode ser usado!
    v1 = v[i] + valor(t,v,i+1,tm-t[i])[0]
    v2 =        valor(t,v,i+1,tm)[0]
    lista1 = valor(t,v,i+1,tm-t[i])[1]  # solução se i na mochila
    lista2 = valor(t,v,i+1,tm)[1]       # solução se i não está na mochila
    if v1>v2:
        return (v1,[i]+lista1)
    return (v2,lista2)
```

Esta função é muito ineficiente, o tempo correspondente é, no pior caso, exponencial uma vez que cada um dos n elementos é seleccionado ou não (2 alternativas). O número de sub-conjuntos de A com $|A| = n$ é 2^n e o tempo de execução da função é $O(2^n)$.

Função sem tabelação, tempo exponencial

```
def valor(t,v,i,n,tm):
  if i>n or tm<=0:
    return 0
  if t[i]>tm:
    res = valor(t,v,i+1,tm)
  else:
    res = max(v[i] + valor(t,v,i+1,tm-t[i]),valor(t,v,i+1,tm))
  return res
```

Função com tabelação, tempo $O(nT)$

```
  Variável val[i][tm] usada para tabelação; inicializada com NAO_DEF

  def valor(t,v,i,n,tm):
    if i>n or tm<=0:
      return 0
(*) if val[i][tm] != NAO_DEF:
      return val[i][tm]
    if t[i]>tm:
      res = valor(t,v,i+1,tm)
    else:
      res = max(v[i] + valor(t,v,i+1,tm-t[i]),valor(t,v,i+1,tm))
(*) val[i][tm] = res
    return res
```

Análise da eficiência da versão com tabelação

Sempre que é efectuada uma chamada `valor(t,v,i,n,tm)`: para estes valores de `i` e `tm` o valor óptimo já foi calculado? Se sim, esse valor é imediatamente devolvido.
Assim, a eficiência pode ser caracterizada pelo número possível de pares $(\texttt{i}, \texttt{tm})$ que tem ordem de grandeza $O(nT)$ onde $T = \texttt{tm}$ é a capacidade da mochila.

Teorema 1 *A utilização da Programação Dinâmica permite resolver o problema "knapsack" (mochila) em tempo $O(nT)$ onde n é o número total de elementos e T é a capacidade da mochila. Qualquer método sem tabelação baseado na consideração de todos os sub-conjuntos possíveis demora um tempo exponencial em n.*

Exercício 1 *É sabido que o problema "knapsack" na sua versão de decisão é completo em NP. Como explica o resultado enunciado no problema anterior?*
A versão de decisão do problema "knapsack" é
INSTÂNCIA: *Um conjunto A de n objectos, $A = \{1, 2, \ldots, n\}$, cada um deles COM um "valor" v_i e um "tamanho" t_i; capacidade T da mochila e valor mínimo a transportar V.*
PERGUNTA: *Existe um sub-conjunto $A \subseteq B$ tal que $\sum_{i \in B} t_i \leq T$ e $\sum_{i \in B} v_i \geq V$?*

Implementação "top-down" da Programação Dinâmica em programas, com tabelação (memoization)

- Vantagem: evita cálculos repetidos
- Desvantagem: "overhead" no tempo de execução