Capítulo 8

Programação Dinâmica: complementos

8.1 Introdução

A Programação Dinâmica é uma técnica de programação para problemas de optimização; baseia-se essencialmente no princípio da optimalidade de Bellman que, em linhas gerais, é o seguinte: qualquer parte de uma solução óptima é óptima. Por exemplo, suponhamos que, numa rede de estradas procuramos o caminho mínimo entre as cidades A e B. Se esse caminho passar pela cidade C, então as partes desse caminho $A \rightarrow C$ e $C \rightarrow B$ são também caminhos mínimos entre A e C e B e C, respectivamente. Assim, a distância mínima entre A e B é dada pela seguinte expressão recursiva

$$d_{A,B} = \min_{C}(d_{A,C} + d_{C,A})$$

Na Programação Dinâmica evita-se que as soluções dos sub-problemas sejam calculadas mais que uma vez. Isso pode ser conseguido de 2 formas

1. Na programação "bottom up" (a forma mais frequente), são calculadas sucessivamente as soluções óptimas dos problemas de tamanho 1, 2,..., até se chegar à solução do problema pretendido.

2. Na programação "top-down": usa-se uma função recursiva para resolver o problema pretendido, de forma análoga ao que se faz na técnica "dividir para conquistar". Essa função chama-se a si mesma para problemas de menores dimensões, *memorizando as soluções dos sub-problemas que vai resolvendo.*

Muitas vezes as funções "top-down" e "bottom up" de uma determinada função baseada na ideia da Programação Dinâmica são, em termos de eficiência muito semelhantes. Contudo, podem existir diferenças: relativamente à versão "bottom up", a versão "top-down" com memorização tem uma desvantagem – o peso das chamadas recursivas – mas pode ter uma vantagem importante, o facto de muitos problemas não chegarem a ser analisados por serem irrelevantes para o problema global.

À técnica de memorização das soluções já encontradas chama-se "tabelação" ("memoization" em Inglês). A tabelação é de certo modo uma outra técnica geral de programação e tem inúmeras aplicações. Entre as questões ligadas à tabelação tem particular importância a escolha de estruturas de dados eficientes para as operações de pesquisa e inserção. Não podemos entrar neste capítulo em mais detalhes sobre a tabelação; limitámo-nos a apresentar a aplicação da tabelação ao algoritmo para a determinação do número de Fibonacci modular.

Algoritmo simplista, tempo exponencial

```
def fibm(n,m):
    if n==0: return 0
    if n==1: return 1
    return (fibm(n-2,m)+fibm(n-1,m)) % m
```

Algoritmo com tabelação.

```
maximo=1000
f = ["NAO_DEF"]*maximo

def fibm(n,m):
    if n==0: return 0
    if n==1: return 1
    if f[n] != "NAO_DEF":
        return f[n]
    res = (fib(n-2,m)+fib(n-1,m)) % m
    f[n]=res
    return res
```

A diferença de tempos de execução entre estas 2 versões é notória: a tabelação converte um algoritmo exponencial num algoritmo polinomial. Sugerimos ao leitor um teste: correr cada uma das versões para os argumentos $\mathtt{n} = 1, 2, \ldots, 50$ (fixando, por exemplo, $\mathtt{m} = 5$).

8.2 Alguns exemplos

Vamos aplicar a ideia essencial da Programação Dinâmica a 3 exemplos concretos: parentização óptima de um produto matricial (página 125), maior sub-sequência comum (página 129) e pro-

blema da mochila (página 134). Em todos estes casos o algoritmo baseado na Programação Dinâmica é muito mais rápido que o algoritmo "simplista".

8.2.1 Parentização óptima de um produto matricial

Como sabemos, o produto matricial é uma operação associativa. Contudo, o correspondente tempo de execução pode depender da forma como se colocam os parêntesis no produto das matrizes. Veremos em breve um exemplo, mas comecemos por caracterizar o *custo* de um produto matricial. Utilizamos o modelo uniforme ver a Secção 2.2 e o exercício 35 na página 54; admitimos que estamos a usar o algoritmo usual da multiplicação de matrizes, e não algoritmos mais elaborados como o de Strassen (página 55).

Definição 13 *Sejam M_1 e M_2 matrizes de dimensões $a \times b$ e $b \times c$ respectivamente. O custo associado ao cálculo do produto M_1M_2 é definido como abc, o número de multiplicações elementares efectuadas.*

Como exemplo, consideremos o produto das matrizes M_1, M_2 e M_3 de dimensões 100×100, 100×10 e 10×2, respectivamente. Os custos associados às 2 parentizações possíveis são

$(M_1M_2)M_3$: $100\,000 + 2\,000 = 102\,000$

$M_1(M_2M_3)$: $20\,000 + 2\,000 = 22\,000$

Assim, é muito mais eficiente calcular o produto na forma $M_1(M_2M_3)$ do que na forma $(M_1M_2)M_3$.

Em termos gerais, o problema é determinar de forma eficiente a parentização óptima de um produto de matrizes, isto é, aquela que leva a um custo de execução mínimo. Note-se que estamos a falar de "eficiência" a 2 níveis: da eficiência do produto matricial (que pretendemos maximizar), e da eficiência da determinação dessa máxima eficiência.

Consideremos um sub-produto do produto $M_1M_2 \ldots M_n$:

$$M_iM_{i+1} \ldots M_{j-1}M_j$$

Em vez de considerar todas as parentizações possíveis deste produto (em número exponencial!), vamos usar a Programação Dinâmica, determinando sucessivamente os custos óptimos dos produtos envolvendo 1, 2,... matrizes consecutivas.

Se $i = j$, temos uma só matriz e o custo é 0. Se $j > i$, o custo óptimo corresponde ao valor de p, com $i \leq p < j$, que minimiza o custo de

$$(M_i \ldots M_p)(M_{p+1} \ldots M_j)$$

onde se assume que os 2 factores devem ser calculados de forma óptima, pelo mesmo processo. Para $i \leq k \leq j$ sejam $d[k] \times d[k+1]$ as dimensões da matriz M_k e seja $c_{a,b}$ o custo mínimo do produto $M_a \ldots M_b$. Então

$$c_{i,j} = \min_{i \leq p < j} \{c_{i,p} + d[i]d[p+1]d[j+1] + c_{p+1,j}\} \tag{8.1}$$

e o valor de p para o qual ocorre o mínimo corresponde a uma escolha óptima do nível mais exterior da parentização. A equação 8.1 permite escrever imediatamente um programa eficiente para definir a parentização óptima. Em linguagem `python` temos

```
 def matmin(d):
 1    infinito = 1E20
 2    m=len(d)-1              # num de matrizes
 3    c     = [[0]*m  for i in range(m)]  # definir mat. bidimensional com 0's
 4    inter = [[-1]*m for i in range(m)]  # definir mat. bidimensional com -1's
 5    for k in range(2,m+1):  # num de matrizes do sub-produto
 6        for i in range(m-k+1):
 7            j=i+k-1
 8            cmin=infinito
 9            for p in range(i,j):
10                custo=c[i][p] + c[p+1][j] + d[i]*d[p+1]*d[j+1]
11                if custo<cmin:
12                    cmin=custo
13                    inter[i][j]=p
14            c[i][j]=cmin
      return (c[0][m-1],inter)
```

Esta função retorna 2 valores: o custo mínimo da parentização `c[0][m-1]` e um vector bi-dimensional `inter` tal que `inter[a][b]` define a parentização exterior óptima, isto é, $M_a \ldots M_b$ deve ser calculado da forma seguinte

$$(M_a \ldots M_p)(M_{p+1} \ldots M_b)$$

onde `p=inter[a][b]`. A seguinte função recursiva utiliza `inter` para imprimir o produto matricial com a parentização óptima

```
# Imprime  Ma ... Mb com a parentização óptima
def expr(inter,a,b):
    p=inter[a][b]
    if a==b:
        printm(a),
        return
    print "(",
    expr(inter,a,p)
    print "*",
    expr(inter,p+1,b)
    print ")",
```

Análise da complexidade

A função `matmin` que determina a parentização óptima tem a eficiência $O(n^3)$. Na verdade, o número de sub-produtos[1] é $O(n^2)$ e para cada um desses sub-produtos a determinação da factorização de topo óptima[2] tem ordem $O(n)$.

Consideremos agora a solução simplista de considerar todas as parentizações possíveis com vista a determinar a óptima. A análise é efectuada nos 2 exercícios seguintes.

Exercício 82 *O produto abcd pode ser parentizada de 5 formas diferentes, nomeadamente* $((ab)c)d$, $(a(bc))d$, $(ab)(cd)$, $a((bc)d)$ *e* $a(b(cd))$. *Mostre que um produto com n factores* $a_1a_2\ldots a_n$ *pode ser parentizado de* $\binom{2(n-1)}{n-1}/n$ *formas diferentes (no exemplo:* $\binom{6}{3}/4 = 20/4 = 5$*).*

Vamos apresentar uma resolução do exercício anterior. O resultado vai ser expresso em termos do número n *de multiplicações*, que é igual ao número de factores menos 1. Isto é, pretendemos mostrar que

$$c_n = \frac{1}{n+1}\binom{2n}{n}$$

Comecemos por determinar o número a_n de modos de colocar $n+1$ factores (por uma ordem qualquer) e de parentizar esse produto. Claramente

$$a_n = c_n(n+1)!$$

Por exemplo, $a_1 = c_1 2! = 2$, correspondendo aos produtos ab e ba. Vamos estabelecer uma recorrência que define a_n em função de a_{n-1}. Seja f_{n+1} o factor "novo" em a_n. Chamemos[3] "arranjos" a cada ordenação parentizada. O novo factor f_{n+1} pode ser colocado nos arranjos correspondentes a a_{n-1} de 2 formas:

- No nível de topo, à esquerda ou à direita: $[f_{n+1} \times (\ldots)]$, $[(\ldots) \times f_{n+1}]$.
- Dentro de cada um dos a_{n-1} arranjos, em cada um dos $n-1$ produtos $A \times B$: de 4 formas:
$$\ldots(f_{n+1} \times A) \times B,\ \ldots(A \times f_{n+1}) \times B,\ \ldots A \times (f_{n+1} \times B),\ \ldots A \times (B \times f_{n+1})$$

[1] Ciclos em `i` e `k` na função `matmin`.
[2] Ciclo em `p` na função `matmin`.
[3] "Arranjos" tem outro significado em combinatória, mas a nossa utilização do termo é local e temporário!

Temos assim, a recorrência

$$\begin{cases} a_1 = 2 \\ a_n = 2a_{n-1} + 4(n-1)a_{n-1} = (4n-2)a_{n-1} \quad \text{para } n \geq 2 \end{cases}$$

Confirmemos o valor de a_2. Temos, da recorrência, $a_2 = 6a_1 = 12$, o que está correcto. A recorrência não é muito difícil de resolver. Temos

$$\begin{aligned} a_n &= (4n-2)a_{n-1} \\ &= (4n-2)(4n-6)a_{n-2} \\ &= (4n-2)(4n-6)\cdots \times 6 \times 2 \\ &= 2^n(2n-1)(2n-3)\cdots \times 3 \times 1 \\ &= 2^n(2n)!/(2n(2n-2)(2n-4)\cdots \times 4 \times 2) \\ &= 2^n(2n)!/(2^n n!) \\ &= (2n)!/n! \end{aligned}$$

donde

$$c_n = \frac{a_n}{(n+1)!} = \frac{(2n)!}{n!(n+1)!} = \frac{1}{n+1}\binom{2n}{n}$$

Nota. Os inteiros c_n são conhecidos como números de Catalan; têm muitas aplicações em Combinatória e em Ciência de Computadores, ver por exemplo,

`http://www.geometer.org/mathcircles/catalan.pdf`

Exercício 83 *Mostre que para $n \geq 4$ é $\binom{2n}{n}/(n+1) \geq 2^{n-1}$.*

Solução. Podemos escrever

$$\binom{2n}{n} \times \frac{1}{n+1} = \frac{(2n) \times \ldots \times (n+1)}{n \times \ldots 1} \times \frac{1}{n+1} = \frac{2n}{n} \times \ldots \times \frac{n}{2} \geq 2^{n-1}$$

uma vez que, para $n \geq 3$ se trata do produto de $n-1$ fracções, todas com valor maior ou igual a 2.

Teorema 25 *A utilização da Programação Dinâmica permite determinar a parentização óptima de um produto de n matrizes em tempo $O(n^3)$. Qualquer método baseado na consideração de todas*

as parentizações possíveis demora um tempo exponencial ou hiper-exponencial em n.

Formulação "top-down" (com tabelação)

O algoritmo que desenvolvemos para a determinar da parentização óptima de um produto de matrizes é do tipo "bottom-up": em primeiro lugar são considerados os sub-produtos de 2 matrizes, depois de 3, de 4, etc. É possível, usando tabelação, conseguir essencialmente a mesma eficiência. É o que mostramos na função seguinte.

```
 1  NAO_DEF = -1
 2
 3  def topdown(d):
 4      m=len(d)-1    # num de matrizes
 5      c     = [[NAO_DEF]*m  for i in range(m)]
 6      inter = [[NAO_DEF]*m for i in range(m)]
 7      return mmtop(d,c,inter,0,m-1)
 8
 9  def mmtop(d,c,inter,i,j):
10      infty = 1E10
11      if i==j:
12          c[i][j]=0
13          return (c,inter)
14      if c[i][j] != NAO_DEF:
15          return (c,inter)
16      cmin=infty
17      for p in range(i,j):
18          custo=mmtop(d,c,inter,i,p)[0][i][p] +\
19                mmtop(d,c,inter,p+1,j)[0][p+1][j] +\
20                d[i]*d[p+1]*d[j+1]
21          if custo<cmin:
22              cmin=custo
23              inter[i][j]=p
24              c[i][j]=cmin
25      return (c,inter)
```

A tabelação (registo de soluções de sub-problemas) ocorre nas linhas `12` e `24`.

Tal como na função "bottom-up", a função `topdown` devolve o par `(c,inter)`. Assim, o custo mínimo global é `topdown(d)[0][0][len(d)-2]`.

Para este problema, e ignorando o peso das chamadas recursivas, a eficiência das versões "bottom-up" e "top-down" é essencialmente idêntica. Usando a versão "top-down" seria possível, e aconselha-se como exercício, evitar eventualmente alguns cálculos de sub-produtos óptimos: se o termo calculado na linha `18` exceder o valor de `cmin`, o cálculo da linha `19` é desnecessário.

8.2.2 Máxima sub-sequência comum

O problema que vamos considerar é a determinação da maior sub-sequência comum a 2 "strings" dados, s e t com comprimentos m e n, respectivamente. É importante notar-se que a sub-sequência

máxima que se procura não é necessariamente constituída por caracteres consecutivos[4], o que torna o problema muito mais difícil; os algoritmos simplistas têm tendência a ser exponenciais, mas uma ideia baseada na Programação Dinâmica vai-nos permitir definir um algoritmo polinomial.

Trata-se de um problema com aplicações a diversas áreas, nomeadamente à genética – por exemplo, análise das semelhanças entre 2 sequências de DNA – e à comparação entre o conteúdo de 2 ficheiros.

Exemplo. Seja $s =$"`abaecc`" e $t =$"`bacbace`". Uma subsequência comum máxima é[5] `abac`, correspondente (por exemplo) aos caracteres sublinhados em

"a b a e c c" "b a c b a c e"

Seja $c_{i,j}$ o comprimento[6] da máxima sub-sequência comum a $s[1..i]$ e a $t[1..j]$. Vamos exprimir $c_{i,j}$ como função de problemas mais pequenos; mais especificamente vamos comparar os últimos caracteres $s[i]$ e $t[j]$:

a. $s[i] \neq t[j]$: neste caso ou $s[i]$ ou $t[j]$ não fazem parte da maior sub-sequência comum. A razão é trivial: $s[i]$ e $t[j]$ são os últimos caracteres de s e t e, como são diferentes, não podem ser (os últimos caracteres) da maior sub-sequência comum. Então $c_{i,j} = \max(c_{i-1,j}, c_{i,j-1})$.

b. $s[i] = t[j]$: uma sub-sequência comum máxima pode ser obtida incluindo este caracter. Isto é, $c_{i,j} = 1 + c_{i-1,j-1}$.

A ideia anterior é a base de um algoritmo eficiente! Podemos considerar pares (i, j) sucessivos, por exemplo, segundo a ordem

$$(1,1),\ (1,2),\ \ldots,(1,m),\ \ldots,\ (2,1),\ (2,2),\ \ldots,(2,m),\ \ldots,\ (n,1),\ (n,2),\ \ldots,(n,m)$$

e determinar o comprimento da maior sub-sequência. Vamos efectuar essa construção para o

[4]Para o caso da pesquisa da maior sub-sequência de caracteres consecutivos, existem algoritmos lineares. Estes algoritmos são generalizáveis a expressões regulares (em vez de sequências fixas) e o leitor poderá encontrar mais informação usando a expressão "string matching" para uma procura na internet.

[5]Há mais 2 que são `abae` `bacc`.

[6]Mais tarde, consideraremos também o problema de determinar a sub-sequência (ou melhor, uma das sub-sequências) que tem esse comprimento, mas para já vamos tratar apenas o problema de determinar o maior comprimento de uma sub-sequência comum.

exemplo dado; a primeira linha da tabela é preenchida com 0's.

	b	a	c	b	a	c	e
	0	0	0	0	0	0	0
a	0	*1*	1	1	1	1	1
b	1	1	1	*2*	2	2	2
a	1	*2*	2	2	*3*	3	3
e	1	2	2	2	3	3	*4*
c	1	2	*3*	3	3	*4*	4
c	1	2	*3*	3	3	*4*	4

A primeira coluna também é muito fácil de preencher. As regras `a` e `b` são usadas para o preenchimento (por linhas) das restantes células:

- (Posições com valores em itálico) se `s[i]=t[j]` (regra `b.`) o valor é 1 mais o conteúdo da célula imediatamente em cima e à esquerda.
- Se `s[i]`$\neq$`t[j]` (regra `b.`), o valor é o maior entre o valor que está em cima e o valor que está à esquerda.

O valor procurado é 4, sublinhado na tabela anterior; é o comprimento da maior sub-sequência comum a "`abaecc`" e "`bacbace`".

Uma sub-sequência comum máxima[7] pode ser obtida a partir do quadro anterior, andando de trás para a frente: verifica-se em que transições foi aumentado de 1 o tamanho da sub-sequência, isto é, quando é que se aplicou a regra `b.`; Essas aplicações correspondem à passagem de um aumento de comprimento da sub-sequência na diagonal descendente e estão marcadas na tabela seguinte (há, como dissemos, outra solução).

	b	a	c	b	a	c	e
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
b	1	1	1	2	2	2	2
a	1	2	2	2	3	3	3
e	1	2	2	2	3	3	4
c	1	2	3	3	3	4	4
c	1	2	3	3	3	4	4

[7] Pode haver mais que uma, mas as regras indicadas correspondem apenas a uma.

A função seguinte, escrita em linguagem `python`, retorna o vector bi-dimensional `ms` que corresponde ao quadro que contruímos atrás. Em `ms[m][n]` está o comprimento da maior sub-sequência comum a `s` e a `t`.

```
# Retorna o comprimento da maior sub-seq comum
# strings 1..len-1  (índice 0 ignorado)
def maxsubseq(s,t):
    m=len(s)   # caracteres 0 1 ... m-1
    n=len(t)   # caracteres 0 1 ... n-1
    ms = [[0]*(n+1) for i in range(m+1)]
    for i in range(1,m):
        for j in range(1,n):
            if s[i] != t[j]:
                ms[i][j] = max(ms[i-1][j],ms[i][j-1])
            else:
                ms[i][j] = 1+ms[i-1][j-1]
    return ms[m-1][n-1]
```

Exercício 84 *Altere o programa anterior por forma a que a função devolva a máxima sub-sequência comum às 2 "strings".* **Sugestão.** *Implemente outra função que recebe* `ms` *e retorna a sub-sequência pedida.*

Na função seguinte calcula-se uma sub-sequência máxima associada a cada prefixo de `s` e de `t`; essas sub-sequências são colocadas no vector bi-dimensional `seq`.

```
# Retorna o comprimento da maior sub-seq. comum e essa sub-seq.
def maxsub(s,t):
    m=len(s)
    n=len(t)
    ms  = [[0]*(n+1) for i in range(m+1)]
    seq = [[""]*(n+1) for i in range(m+1)]
    for i in range(1,m):
        for j in range(1,n):
          | if s[i] != t[j]:
          |     if ms[i-1][j] >= ms[i][j-1]:
          |         ms[i][j]  = ms[i-1][j]
          |         seq[i][j] = seq[i-1][j]
          |     else:
          |         ms[i][j]  = ms[i][j-1]
          |         seq[i][j] = seq[i][j-1]
          | else:
          |     ms[i][j]  = 1+ms[i-1][j-1]
          |     seq[i][j] = seq[i-1][j-1]+s[i] # concat. de strings
        print ms[i]
    return seq[m-1][n-1]
```

Análise da eficiência

Analisando o programa da página 132, verifica-se que o seu tempo de execução é $O(mn)$; basta para isso verificar que a parte do programa marcada com "|" é executada mn vezes. Assim, temos

Teorema 26 *Sejam m e n os comprimento das "strings" s e t. A utilização da Programação Dinâmica permite resolver o problema da maior sub-sequência comum às 2 "strings" em tempo $O(mn)$. Qualquer método baseado na procura de cada sub-sequência da primeira "string" na segunda "string" demora um tempo exponencial em m.*

Notas complementares sobre o problema da maior sub-sequência comum

O problema da maior sub-sequência comum é equivalente ao problema da distância de edição mínima ("minimum edit distance"): dadas 2 "strings" s e t, qual o número mínimo de operações de edição que são necessárias para converter s em t? As operações de edição permitidas são

- Inserir um caracter.
- Eliminar um caracter

A modificação de s em t com um número mínimo de operações de edição pode ser obtida com

$$\boxed{(m - \texttt{maxsub}(s,t)) \text{ eliminações seguidas de} \ldots (n - \texttt{maxsub}(s,t)) \text{ inserções}}$$

num total de $m + n - 2 \times \texttt{maxsub}(s,t)$ operações elementares de edição.

Em biologia computacional usa-se frequentemente a noção mais geral de "alinhamento de sequências", mas as técnicas que descrevemos baseadas na Programação Dinâmica são também aplicáveis nesse caso.

Versão "top-down" e tabelação

Tal como noutras aplicações da Programação Dinâmica, podemos, com a ajuda da tabelação, implementar uma versão "top-down" da função que determina o comprimento da sub-sequência mais longa (ou a própria sub-sequência).

Vejamos, em primeiro lugar, uma descrição em pseudo-código de uma versão "top-down" sem tabelação, extremamente ineficiente:

```
   def maxsubseq(s,i,t,j):
      if i==0 or j==0:
        return 0
      if s[i] == t[j]:
        r = 1 + maxsubseq(s,i-1,t,j-1)
      else:
        r = max(maxsubseq(s,i-1,t,j),maxsubseq(s,i,t,j-1))
      return r
```

A chamada exterior a esta função é da forma `maxsubseq(s,len(s),t,len(t))`.

Vamos agora transformar este programa, introduzindo a tabelação. O vector `val[i][j]`, inicializado com `NAO_DEF`, vai conter o comprimento da maior sub-sequência comum a `s[1..i]` e `t[1..j]`. As linhas marcadas com `(*)` correspondem à tabelação.

```
   def maxsubseq(s,i,t,j):
      if i==0 ot j==0:
        return 0
(*)   if val[i][j] != NAO_DEF:
        return val[i][j]
      if s[i] == t[j]:
        r = 1 + maxsubseq(s,i-1,t,j-1)
      else:
        r = max(maxsubseq(s,i-1,t,j),maxsubseq(s,i,t,j-1))
(*)   val[i][j] = r
      return r
```

8.2.3 Problema da mochila ("knapsack problem")

Suponhamos que temos um conjunto A de n objectos, $A = \{1, 2, \ldots, n\}$ e que cada um deles tem um "valor" v_i e um "tamanho" t_i. Dispomos de uma mochila com um tamanho T. Pretende-se determinar um conjunto $B \subseteq A$ de objectos que cabe na mochila e maximiza o valor "transportado". Por outras palavras, B deve satisfazer

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

Aquilo que se entende por "valor" ou "tamanho" depende evidentemente do caso concreto em consideração. Por exemplo, podemos ter a situação

$$\begin{cases} A: & \text{problemas propostos ao aluno num exame} \\ B: & \text{problemas que o aluno deverá resolver} \\ t_i: & \text{tempo que o aluno demora a fazer o problema } i \\ v_i: & \text{cotação do problema } i \\ T: & \text{tempo total do exame} \end{cases}$$

(supõe-se que cada resposta é valorizada com 100% ou com 0%)

Vejamos um exemplo. O tempo total do exame é 16 horas (!) e temos

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

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])
Comentário: o índice 0 das listas não é usado

def valor(t,v,i,tm):
    if i==len(t) or t<=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]
    lista2 = valor(t,v,i+1,tm)[1]
    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 $O(2^n)$.

Como obter uma função mais eficiente? Vamos usar uma variável para tabelar as computações, isto é, para guardar os valores já calculados. Para simplificar um pouco a escrita da função, supomos que esta retorna apenas o valor óptimo. Em pseudo-código, a função sem tabelação, é

```
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
```

E, com tabelação:

```
   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

Seja n o número total de elementos da lista e T o espaço da mochila (ou tempo total do exame...). Sempre que é efectuada uma chamada `valor(t,v,i,n,tm)`, se para estes valores de `i` e `tm` o valor óptimo já foi calculado, 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 $O(nT)$.

Teorema 27 *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 85 *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 $\sum_{i \in B} v_i \geq V$?*

8.3 Comentário final

Estudamos em detalhe a aplicação da Programação Dinâmica a 3 exemplos específicos: parentização de matrizes, máxima sub-sequência comum e problema da mochilas. Mas a utilidade deste "método" é muito mais vasta; o leitor poderá consultar por exemplo [2] para ficar com uma melhor ideia do campo de aplicação da Programação Dinâmica.