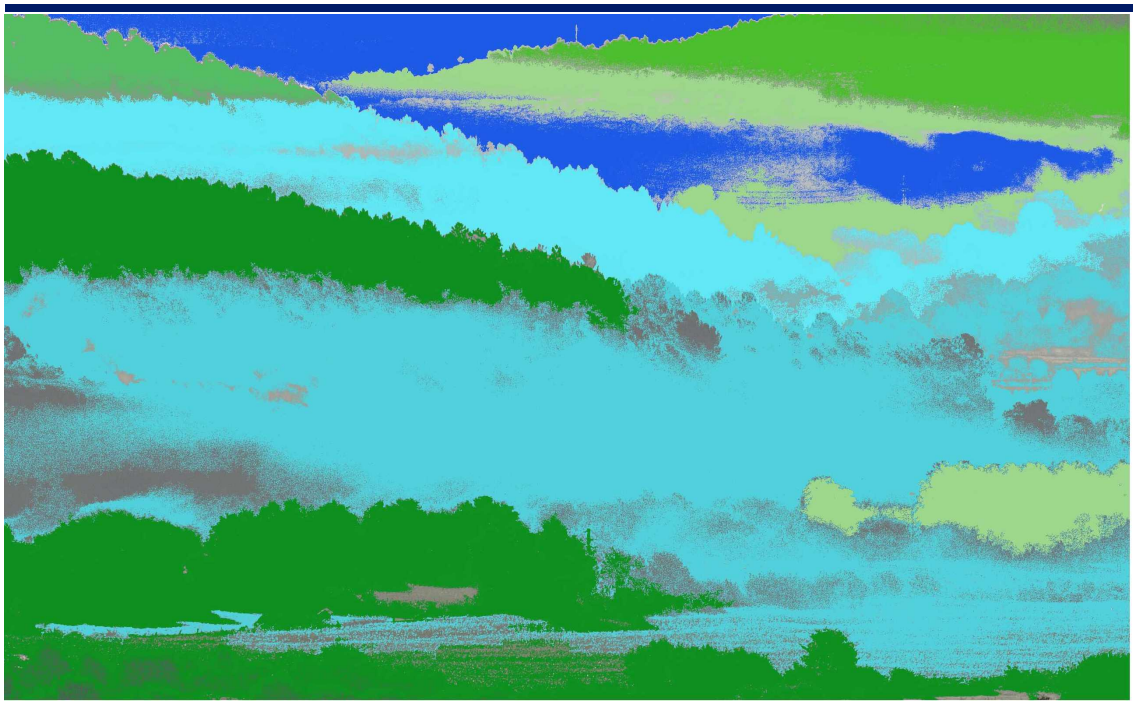
Tópicos Avançados em Algoritmos

Armando Matos



2008

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Conteúdo

Capítulo 0

Introdução

Nesta publicação reunimos um conjunto de notas informais das aulas teóricas da disciplina de "Tópicos Avançados em Algoritmos"[1]. Incluimos também diversos exercícios propostos nas aulas teóricas.

Uma grande parte da Ciência de Computadores consiste no estudo dos algoritmos – projecto ("design"), prova da correcção, análise da eficiência e implementação[2]. Há muitos, muitos, algoritmos nas mais diversas áreas: algoritmos de ordenação e pesquisa, algoritmos para procura de padrões em textos ("pattern matching") algoritmos de compressão de ficheiros, algoritmos numéricos, algoritmos geométricos, algoritmos para análise léxica ou sintática, algoritmos para "garbage collection", algoritmos para problemas relacionados com grafos, algoritmos usados na Teoria dos Grupos... a lista quase não tem fim. Contudo, vale a pena estudarmos algoritmos específicos: quando se compreende bem ou se inventa um algoritmo concreto está-se muitas vezes a utilizar uma ideia ou um conjunto de ideias que são aplicáveis com muito mais generalidade.

Nesta publicação estudaremos os seguintes temas relacionados com algoritmos:

- Algumas técnicas de análise da eficiência de algoritmos.
- Algoritmos específicos.
- Algumas técnicas genéricas aplicáveis em muitos problemas: "dividir para conquistar" e Programação Dinâmica.
- Minorantes de complexidade.

[1] Anos de 2008 e 2009, Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto.

[2] O projecto, a prova da correcção e a análise da eficiência estão intimamente relacionados; devem efectuar-se "em paralelo".

Este livro está organizado daseguinte forma. No próximo capítulo apresentam-se alguns elementos de análise de algoritmos, sendo especialmente estudadas as ordens de grandeza das funções e alguns métodos de solução de recorrências. O Capítulo 2 trata dos modelos de computação e do tempo de execução dos algoritmos, estudando-se em especial a análise amortizada do tempo de execução. O esquema "dividir para conquistar", bem como a solução das recorrências que lhe estão associadas são estudados no Capítulo 3; esse esquema é aplicado a algoritmos de multiplicação de inteiros e de matrizes grandes. No capítulo seguinte são considerados os algoritmos aleatorizados, isto é, que têm acesso a uma fonte de números aleatórios (ou pseudo-aleatórios); em particular é estudado o "quick-sort" aleatorizado e o algoritmo de primalidade Rabin-Miller. As classes de complexidade "aleatorizadas" são também estudadas. O Capítulo 5 trata de alguns algoritmos relacionados com o problema da ordenação, sendo considerados métodos de ordenação aplicáveis a universos "pequenos", o "radix-sort" e um algoritmo eficiente de determinação da mediana. Os "circuitos" como modelos de computação são mencionados no Capítulo 6, sendo estudadas em algum pormenor as redes de ordenação. No Capítulo 7 são consideradas 2 questões relacionadas com os métodos de "hash": o "hash" universal e o "hash" perfeito. Algumas aplicações típicas da Programação Dinâmica – parentização óptima de um produto matricial, máxima sub-sequência comum e o problema da mochila ("knapsack problem") – são considerados em algum pormenor no Capítulo 8. O capítulo seguinte trata do importante algoritmo FFT ("Fast Fourier Transform"); este algoritmo é estudado do ponto de vista da conversão entre 2 representações dos polinómios – representação por coeficientes e representação por valores em determinados pontos. A Teoria da Informação é aplicada no Capítulo 10 à determinação de minorantes de complexidade de algoritmos de ordenação e pesquisa.

Pré-requesitos. Para a boa compreensão destes apontamentos é necessário : (i) ter alguma maturidade matemática, em particular na área da matemática discreta, (ii) ter conhecimentos mínimos da teoria das probabilidades, (iii) conhecer os principais algoritmos de ordenação e respectiva eficiência, (iv) ter alguma experiência de programação numa linguagem como, por exemplo, o C, o C++, o Python ou até o Java, e ter capacidade de implementar nessa linguagem algoritmos descritos numa linguagem informal, (v) conhecer os fundamentos da teoria dos problemas completos em NP.

Os exercícios são parte integrante deste curso.

O leitor deverá fazer um esforço sério para os resolver!

Capítulo 1

Preliminares: fundamentos da análise de algoritmos

Este capítulo é uma exposição sumária de alguns preliminares relativos à análise de algoritmos. Depois de referirmos os modelos de computação normalmente utilizados e de mencionar os principais recursos utilizados durante a execução de um programa (tempo e espaço), estudaremos 2 aspectos fundamentais para a análise de algoritmos: ordens de grandeza das funções e solução de recorrências. Os conceitos e os resultados apresentados neste capítulo são fundamentais para a análise da eficiência dos algoritmos apresentados neste curso.

1.1 Eficiência dos algoritmos

1.1.1 Eficiência dos algoritmos: duas análises

1. Pormenorizada: mais exacta e directa mas em geral menos útil
   - Expressa em segundos.
   - Resultado da avaliação da eficiência (por exemplo: tempo gasto): único e dependente da velocidade e características do processador.
2. Através de ordens de grandeza: ignora as constantes multiplicativas e é uma análise assintótica ←
   - Expressa em *ordens de grandeza*

– Resultado da avaliação da eficiência: paramétrico (uma função do comprimento dos dados) e independente da velocidade e características do processador.

Nesta disciplina usaremos a análise 2, "através de ordens de grandeza"!

1.1.2 Recursos e modelos de computação

Em que termos é medida a "eficiência de um algoritmo"?

Resposta: pela quantidade de recursos gastos durante a sua execução como função do *comprimento dos dados*.

Recursos? O que são?

Resposta: tempo (o mais usual) e espaço (memória)

Modelos de computação mais usados:

1. Máquinas de Turing (MT)
2. Computadores de registos ("Random Access Machines")

Tempo e espaço, o que são?

Nas MT

1. Tempo: número de transições da MT durante a computação
2. Espaço: número de células visitadas pela cabeça da MT durante a computação

1.1.3 Pior caso e caso médio

Eficiência do algoritmo $\boxed{\text{A}}$ como função de quê?

Resposta: Do comprimento dos dados.

$$x \longrightarrow \boxed{\text{A}} \longrightarrow y$$

Seja $n = |x|$

Tempo de execução: depende dos dados, $t(x)$

Simplifica-se: como função do comprimento dos dados

$$t = f(n) \text{ onde } n = |x|$$

Mas... isto está errado, o tempo de execução é função de x e não de $n = |x|$!

Temos que escolher um tempo de entre todos os tempos $t(x)$

com $n = |x|$

Hipóteses mais frequentes:

1. Pior caso: $t(n) = \max\{t(x) : |x| = n\}$

2. Caso médio: $t(n) = E\{t(x) : |x| = n\}$ onde se assume uma determinada distribuição probabilística dos dados x de comprimento n; x e $t(x)$ são variáveis aleatórias. Significado de E: "valor médio".

1.1.4 Recorrências, pior caso e caso médio, um exemplo

Seja o programa

```
//-- Dado: vector v[] com n elementos, v[0,1,...,n-1]
1    int i, m
2    m=0;
3    for i=1 to n-1
4      if v[i] > v[m] then
5         m=i
6    print m
```

É fácil ver que o tempo de execução é majorado por

$$\alpha n + \beta$$

onde α e β são constantes, ver por exemplo Knuth, "The Art of Computer Programming", vol. 1.

Concentremo-nos na seguinte questão:

Quantas vezes é executada a atribuição da linha `5`?

Seja a esse número de vezes.

- Mínimo: $a = 0$ vezes
- Máximo: $a = n - 1$ vezes
- Médio: $E(a) =$???

Hipóteses sobre a distribuição probabilística dos dados

1. Todos os elementos de `v[]` são diferentes. Note que os valores do vector são irrelevantes, só é importante a sua "ordem", por exemplo, o comportamento para `v=[4,1,2,5,3]` e para `v=[41,2,5,55,30]` é idêntico.

2. Qualquer das $n!$ permutações é igualmente provável.

Calculemos alguns valores de prob $a = i$

- prob $a = 0$, probabilidade de ser $a = 0$ é $1/n = (n-1)!/n!$ = probabilidade de `v[0]` ser o maior de todos.
- prob $a = n - 1 =$ é $1/n!$ pois só numa das $n!$ permutações isso acontece.

Teremos que calcular a média

$$E(a) = 0 \times \text{prob } a = 0 + 1 \times \text{prob } a = 1 + \cdots + (n-1) \times \text{prob } a = n - 1$$

Vamos calcular $E(a)$ como uma função de n, seja $E(n)$. Temos uma recorrência:

1. $E(1) = 0$
2. Imaginemos que o vector tem $n + 1$ elementos e calculemos como é que $E(n + 1)$ depende de $E(n)$. Temos

$$E(n+1) = E(n) + 1 \times \text{prob o último elemento ser o maior}$$

Ou seja

$$\begin{cases} E(1) = 0 \\ E(n+1) = E(n) + 1/n \end{cases}$$

A solução desta recorrência é fácil

$$E(n) = 0 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} = \left(\sum_{i=1}^{n-1} \frac{1}{i} \right) - 1$$

Por exemplo, para $n = 1000$ temos

- Mínimo $= 0$
- Máximo $= 999$
- Média $E(n) \approx 6.49$

Mas isto não é uma "forma fechada"!

Podemos determinar uma forma fechada?

Ou... talvez uma boa aproximação?

Exercício 1 *Confirme o resultado anterior para um vector de 3 elementos distintos, estudando as* $6 = 3!$ *situações possíveis:*

`v[0]<v[1]<v[2],...v[2]<v[1]<v[0]`

(Fim do exemplo)

Investigar o caso geral! $E(n) = \ldots$?

1.2 Ordens de grandeza

Como "ignorar constantes e comportamento inicial"?

Resposta: **usando ordens de grandeza**.

1.2.1 Majoração, minoração e ordem exacta

Ordens de grandeza (ideia)

1. Majoração de $f(n)$ por $g(n)$:
   $f(n)$ é de ordem $O(g(n))$ – ou $f(n) \in O(g(n))$.
   Para valores suficientemente grandes de n é $f(n) \leq kg(n)$ onde k é uma constante real positiva.

2. Minoração de $f(n)$ por $g(n)$:
   $f(n)$ é de ordem $\Omega(g(n))$ – ou $f(n) \in \Omega(g(n))$.
   Para valores suficientemente grandes de n é $f(n) \geq kg(n)$ onde k é uma constante real positiva.

3. Ordem de grandeza exacta:
   $f(n)$ é de ordem $\Theta(g(n))$ – ou $f(n) \in \Theta(g(n))$.
   Para valores suficientemente grandes de n é $k_1 g(n) \leq f(n) \leq k_2 g(n)$ onde k_1 e k_2 são constantes reais positivas.

As fefinições formais das ordens de grandeza $O()$, $\Omega()$ e Θ são as seguintes.

Definições (como conjuntos):

Sejam f e g funções de $\mathbb{N}$ em $\mathbb{R}$. Seja $\mathbb{R}^+$ o conjunto dos reais positivos.

– Majoração: $g(n)$ é $O(f(n))$, f é majorante ("upper bound") de g

$$O(f(n)) = \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k \in \mathbb{R}^+, \ \forall n \geq n_0 : \ \ g(n) \leq kf(n)\}$$

– Minoração: $g(n)$ é $\Omega(f(n))$, f é minorante ("lower bound") de g

$$\Omega(f(n)) = \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k \in \mathbb{R}^+, \ \forall n \geq n_0 : \ \ g(n) \geq kf(n)\}$$

– $g(n)$ é $\Theta(f(n))$, f é da ordem de grandeza exacta de g

$$\begin{aligned}\Theta(f(n)) \quad = \quad & \{g(n) \ : \ \exists n_0 \in \mathbb{N}, \exists k_1, k_2 \in \mathbb{R}^+, \\ & \forall n \geq n_0 : \ \ k_1 f(n) \leq g(n) \leq k_2 f(n)\}\end{aligned}$$

Exercício 2 *Compare estas definições com as da página anterior; dê um significado a "suficientemente grande" por forma a que as definições coincidam.*

As 3 ordens de grandeza – notas

– Note-se que $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$ são conjuntos. Quando dizemos $n^2 + 3$ é de ordem $O(n^3)$ queremos dizer $n^2 + 3 \in O(n^3)$.

– Para que serve o n_0, porquê apenas para valores de n suficientemente grandes?

  – O domínio de $f(n)$ pode não ser $\mathbb{N}$. Por exemplo, diz-se que $\log n^2$ é de ordem $O(n^2)$ embora a função não esteja definida para $n = 0$.

  – $f(n)$ pode tomar valores nulos ou negativos...

– A noção de ordem pode ser estendida para funções $\mathbb{R}_0^+ \to \mathbb{R}^+$.

– Gráficos ilustrativos (no quadro!)

Exercício 3 *Verdadeiro ou falso? Justifique*

1. $\log n \in O(n)$
2. $n \in O(\log n)$
3. $2n^2$ *é de ordem* $O(n^2)$

4. $2n^2$ *é de ordem* $O(n^3)$
5. $\Omega(n) = \Omega(3n)$ *(igualdade de conjuntos)*
6. *4 é* $O(1)$
7. $O(n^2) \subseteq \Omega(n^2)$

Exercício 4 *Mostre que a seguinte função*

$$f(n) = \begin{cases} 2n & \text{se } n \text{ é par} \\ 2 & \text{se } n \text{ é ímpar} \end{cases}$$

é de ordem $O(n)$ *mas não de ordem* $\Theta(n)$.

Exercício 5

Porque é que usualmente não se diz que uma determinada função é de ordem $O(2n)$ *ou* $\Omega(3n)$ *ou* $\Theta(4n)$*? O coeficiente que se usa é sempre 1; assim, diz-se ordens* $O(n)$, $\Omega(n)$ *e* $\Theta(n)$ *respectivamente.*

Exercício 6

Considere a seguinte relação binária entre funções totais de $\mathbb{N}$ *em* $\mathbb{R}^+$*:* fRg *sse* $f(n)$ *é de ordem* $O(g(n))$*. Averigue se a relação é simétrica. Repita o exercício para a ordem de grandeza* Θ.

Exercício 7

Diz-se que $f(n)$ *é de ordem* $o(g(n))$*, quando*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Por exemplo, $1/(n^2 \log n)$ *é* $o(1/n^2)$*. Consegue encontrar alguma relação matemática entre esta definição e as "nossas" ordens de grandeza?*

1.2.2 Tempo de execução em algoritmos de ordenação e pesquisa

Para exprimir o tempo de execução usa-se muitas vezes

- o número de comparações envolvendo elementos do vector em questão

Exemplo

```
    //-- ordena v[0..n-1] pelo método da selecção do mínimo
    int i,j,m,t;
1   for i=0 to n-2
2   | m=i;
3   | for j=i+1 to n-1
4   |   if v[j]<v[m]
5   |    m=j
6   | t=v[i]; v[i]=v[j]; v[j]=m;
    // v está ordenado!
```

Exercício 8 *Quantas comparações* `v[j]<v[m]` *são efectuadas (linha* `4`*)? Exprima a sua resposta como uma função de* `n`*.*

Exemplo

```
    //-- "merge" de a[0..m-1] e b[0..n-1] (já ordenados)
    //   em v[]
    //-- Ex: a=[1,4,5,8], b=[2,3,4] --> v=[1,2,3,4,4,5,8]
1   int i=0, j=0, k=0;
2   while i<m && j<n
3     if a[i]<b[j]
4       v[k]=a[i]; i=i+1; k=k+1;
5     else
6       v[k]=b[j]; j=j+1; k=k+1;
7   while i<m: v[k]=a[i]; i=i+1; k=k+1; // só um destes ciclos
8   while j<n: v[k]=b[j]; j=j+1; k=k+1; // é executado. Porquê?
```

(Note que um dos ciclos finais (linhas `7` e `8` é "executado" 0 vezes)

Exercício 9 *Determine o número de comparações* `a[i]<b[j]` *efectuadas (na linha* `3`*)*

1. *Valor mínimo (pode supor* `m<n`*).*
2. *Valor máximo.* RESPOSTA: $m + n - 1$.

Exercício 10 *Dê um exemplo de um algoritmo de ordenação em que o número de comparações $c(n)$ envolvendo elementos do vector não é da mesma ordem que o tempo de execução do algoritmo. Assim, neste caso, $c(n)$ não "exprime" correctamente o tempo de execução do algoritmo.*

1.3 Solução de recorrências

1.3.1 Exemplos de funções definidas através de recorrências – definições indutivas

Factorial

$$n! = \begin{cases} 1 & (\text{se } n = 0) \\ n(n-1)! & (\text{se } n \geq 1) \end{cases} \qquad \text{ou...} \qquad \begin{cases} f(0) = 1 & (\text{se } n = 0) \\ f(n) = nf(n-1) & (\text{se } n \geq 1) \end{cases}$$

Fibonacci

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad (\text{se } n \geq 2) \end{cases}$$

Que será?

$$\begin{cases} f(0) = 0 \\ f(n+1) = f(n) + 3n^2 + 3n + 1 \quad (\text{se } n \geq 0) \end{cases}$$

Que será?

$$\begin{cases} f(1) = 0 \\ f(2n) = 2f(n) + 2n \quad (\text{para } n \geq 1) \end{cases}$$

Nota: só está definido quando n é potência de 2.

1.3.2 O que é uma recorrência

O que é uma recorrência?

Resposta: um método de definir sucessões.

Uma sucessão $f : \mathbb{N} \to \mathbb{R}$ pode ser definida por um conjunto finito de equações dos seguintes tipos:

- Equações fronteira: $f(k) = c$ onde k é uma constante inteira não negativa e c é uma constante real.

  Por exemplo: $f(2) = 1$.

- Equações gerais:

  $$f(n) = \exp(\cdots) \text{ para } n\ldots$$

  onde n é uma variável inteira e $\exp(\cdots)$ é uma expressão que pode envolver valores de $f(i)$ para $i < n$.

  Por exemplo: $f(n) = f(n-1) + f(n-2)$ para $n \geq 2$.

As equações gerais e fronteira devem definir univocamente todos os valores de $f(i)$. Normalmente existe apenas uma equação geral.

Há uma relação muito próxima entre

recorrências | definições indutivas

demonstração por indução finita

Há uma relação próxima entre

recorrências $\leftrightarrow$ eq. diferenciais

Por vezes a solução é directa...

Por vezes a solução é directa...

Um exemplo:

$$\begin{cases} f(0) = 1 \\ f(n+1) = f(n) + n \end{cases}$$

Nota. "$f(n+1) = f(n) + n$ para $n \geq 0$" é equivalente a "$f(n) = (n-1) + f(n-1)$ para $n \geq 1$" (mudança de variável).

Temos

$$\begin{aligned} f(n) &= (n-1) + f(n-1) \\ &= (n-1) + (n-2) + f(n-2) \\ &= \cdots \\ &= ((n-1) + (n-2) + \cdots + 1 + 0) + 1 \\ &= n(n-1)/2 + 1 \end{aligned}$$

Portanto a solução é

$$f(n) = \frac{n^2 - n + 2}{2}$$

Neste caso a solução directa foi possível porque sabemos somar progressões aritméticas.

Nota. O aluno deve conhecer as fórmulas das somas de progressões aritméticas e geométricas (ou saber deduzi-las...)

Um exercício

Exercício 11 *Determine directamente a solução da seguinte recorrência*

$$\begin{cases} f_1 = 2 \\ f_{n+1} = 3f_n \end{cases}$$

Observações

- *Domínio.* Nem sempre uma recorrência define uma sucessão para todos os valores de n. Por exemplo, em algoritmos de ordenação pode simplificar-se a análise se se supuser que o número de elementos é uma potência de 2.
- *Existência.* para muitas recorrências não é conhecida – e possivelmente não existe – uma forma fechada para a sua solução.
- *Majoração.* Por vezes a solução de uma recorrência é muito difícil ou impossível, mas pode por vezes a partir dela definir outra mais simples cuja solução majora a função caracterizada pela recorrência; isso pode ser suficiente para, por exemplo, conhecer uma "ordem de

grandeza" da solução.

Isso pode acontecer, por exemplo, quando a função definida pela recorrência é o tempo de execução de um algoritmo.

1.3.3 Método Tabelar → suspeitar → demonstrar

O "método"

1. Tabelar: usando a recorrência tabelamos os primeiros valores da função; para ajudar no passo seguinte podem tabelar-se outras funções como n^2, 2^n, $\log n$, etc.

2. Suspeitar: eventualmente a tabela construída no passo anterior pode levar-nos a suspeitar que a solução é uma determinada função de n, seja $f(n)$.

3. Demonstrar: provamos – usualmente usando o método da indução finita – que $f(n)$ é de facto (se for!) a solução da recorrência.

Claro que se não conseguirmos avançar no passo 3. pode a suspeita estar errada e há que retroceder ao passo 2.

Um exemplo simples

$$\begin{cases} a_0 = 0 \\ a_{n+1} = a_n + 2n \end{cases}$$

Tabelar

n	a_n
0	0
1	0
2	2
3	6
4	12
5	20

Qual parece ser a solução?

Suspeitar

... coloquemos a coluna n^2

n	n^2	a_n
0	0	0
1	1	0
2	4	2
3	9	6
4	16	12
5	25	20

Agora, a suspeita é fácil!

$$f(n) = n^2 - n? \qquad \text{(suspeita)}$$

Demonstrar

O quê?

Teorema 1 *A solução da recorrência é* $n^2 - n$.

Dem. Por indução em n.

I. Caso $n = 0$: $0^2 - 0 = 0$. E da recorrência é $a_0 = 0$. $\surd$

II. Demonstrar que $\boxed{a_n = n^2 - n \;\Rightarrow a_{n+1} = (n+1)^2 - (n+1)}$.

Temos

$$\begin{array}{rcll} a_{n+1} & = & a_n + 2n & \text{(da recorrência)} \\ & = & (n^2 - n) + 2n & \text{(hipótese indutiva)} \\ & = & (n+1)^2 - (n+1) & \text{(contas simples!)} \end{array}$$

. . . e está demonstrada a implicação. □

Exercício 12 *O "mergesort" é um método de ordenação muito eficiente, mesmo no pior caso. Descrição, supondo que n é uma potência de 2:*

`mergesort(v[0...n-1],n):`

1. *Se* $n = 1$*: nada se faz*
2. *Se* $n \geq 2$*:*
   - *(a)* $\texttt{mergesort}(\texttt{v}[0 \cdots \texttt{n}/2 - 1], \texttt{n}/2)$
   - *(b)* $\texttt{mergesort}(\texttt{v}[\texttt{n}/2 \cdots \texttt{n} - 1], \texttt{n}/2)$
   - *(c)* $\texttt{merge}(\texttt{v}[0 \cdots \texttt{n}/2 - 1], \texttt{v}[\texttt{n}/2 \cdots \texttt{n} - 1]) \longrightarrow \texttt{v}[0 \cdots \texttt{n} - 1]$

Já definimos o que é o `merge`*, ver página 20.*

1. *Ilustre a execução do mergesort para o vector*

$$v[\,] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 9 & 7 & 8 & 5 & 1 & 3 & 6 & 2 \\ \hline \end{array}$$

2. *Da definição resulta a seguinte recorrência para um majorante do número de comparações efectuadas (são todas efectuadas nas chamadas de* `merge`*); use o máximo indicado na página 20 com* $m \to n/2$, $n \to n/2$*:*

Resposta: _____

3. *Resolva a recorrência pelo método "tabelar / suspeitar / demonstrar".*

1.3.4 Método das diferenças finitas constantes

Definição 1 *Seja* a_n, $n \in \mathbb{N}$, *uma sucessão. A diferença finita (ou simplesmente diferença) de* $1^{\underline{a}}$ *ordem de* a_n *é a sucessão*

$$D_n^1 = a_{n+1} - a_n \qquad \textit{(para } n \geq 0\textit{)}$$

Para $i \geq 2$ *a diferença de ordem* i *de* a_n *é a sucessão*

$$D_n^i = D_{n+1}^{i-1} - D_n^{i-1}$$

Exemplo. Para a sucessão a_n definida pela recorrência

$$\begin{cases} a_0 = 1 \\ a_n = a_{n-1} + n^2 \quad \text{para } n \geq 1 \end{cases}$$

as diferenças de ordem 1 e 2 são

n	a_n	D_n^1	D_n^2
0	1	1	3
1	2	4	5
2	6	9	$\cdots$
3	15	$\cdots$	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

Teorema 2 *Se as diferenças finitas de ordem* m *da sucessão* a_n *são uma constante não nula, a solução da recorrência é um polinómio de grau* m*, isto é*

$$a_n = c_m n^m + c_{m-1} n^{m-1} + \cdots + c_1 n + c_0$$

com $c_m \neq 0$.

Nota. A *diferença* de ordem m é como que a imagem discreta, isto é, em termos das funções de $\mathbb{N}$ em $\mathbb{R}$, da *derivada* de ordem m. No domínio "contínuo" o Teorema anterior tem a seguinte "versão": se a derivada de ordem m de uma função $f(x)$ é uma constante não nula, então $f(x)$ é um polinómio de grau m em x.

Método 1 da solução da recorrência

1. Provar que, para um determinado inteiro m, as diferenças finitas de ordem m são uma constante não nula.
2. Usar o método dos coeficientes indeterminados ($m+1$ valores de a_n) para determinar os coeficientes do polinómio.

Método 2 da solução da recorrência

1. Tabelar os primeiros valores de: a_n, D_n^1,..., D_n^m.
2. Verificar que (se) D_n^m aparenta ser uma constante.
3. Usar o método dos coeficientes indeterminados ($m+1$ valores de a_n) para determinar os coeficientes do polinómio.
4. Provar que o polinómio obtido é solução da recorrência.

O seguinte resultado permite em alguns casos a aplicação directa do método das diferenças finitas constantes.

Teorema 3 *A solução de uma recorrência da forma $t_0 = a$, $t_{n+1} = t_n + p(n)$ onde a é uma constante e $p(n)$ um polinómio de grau d é um polinómio de grau $d+1$.*

Exercício 13 *(i) Continuar a tabela do exemplo anterior (pág. 26), mostrando que D_n^3 aparenta ser uma constante positiva. Determinar a solução da recorrência (método 2). (ii) Resolver directamente a recorrência usando o Teorema 3.*

Exercício 14 *Determinar uma fórmula fechada para a soma dos primeiros n quadrados,*

$$S_n = \sum_{i=1}^{n} i^2$$

Use o exercício anterior.

1.3.5 Método da mudança de variável

Vamos usar um exemplo.

Seja a recorrência

$$\begin{cases} a(1) = 1 \\ a(2n) = a(n) + 1 \end{cases}$$

Note que $a(n)$ só fica definido quando n é potência de 2, $n = 2^p$, $p \geq 0$.

É natural efectuar a mudança de variável $n = 2^p$ (ou $p = \log n$). Isto é vamos representar $a(b)$ por uma outra recorrência $b(p)$ sendo

$$a(n) \equiv b(p)$$

Fica

$$\begin{cases} b(0) = 1 \\ a(p+1) = b(p) + 1 \end{cases}$$

cuja solução é muito simples, $b(p) = p + 1$, ou seja, em termos de n:

$$a(n) = \log n + 1$$

1.3.6 Método da equação característica homogénea

Estudamos agora a solução de recorrências em f_n com uma única equação geral da forma

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = 0$$

onde k é uma constante e $a_0 \neq 0$.

Exercício 15 *Mostre que as equação geral da sequência de Fibonacci é da forma indicada.*

Se experimentarmos soluções da forma $f_n = r^n$ vemos que r deve satisfazer a equação (dita característica)

$$a_0 r^k + a_1 r^{k-1} + \cdots + a_r = 0$$

Teorema 4 *Se $f_n = f_1(n)$ e $f_n = f_2(n)$ são soluções da equação $a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = 0$, então, sendo α e β quaisquer constantes, $\alpha f_1(n) + \beta f_2(n)$ também é solução dessa equação.*

Exercício 16 *Demonstre este resultado.*

Teorema 5 *Se as raízes da equação característica são todas distintas (reais ou complexas), sejam $r_1, \ldots, r_k$, a solução da equação geral é exactamente constituída pelas sucessões da forma*

$$f_n = \alpha_1 {r_1}^n + \alpha_2 {r_2}^n + \cdots + \alpha_k {r_k}^n$$

onde $\alpha_1, \alpha_2, \ldots,$ e α_k são constantes arbitrárias.

Nota. Entre as soluções possíveis está $f(n) = 0$ (solução identicamente nula) que se obtém com $\alpha_1 = \alpha_2 = \ldots = \alpha_k = 0$.

1.3.7 Equação característica homogénea – raízes distintas

Método:

1. Determine a equação característica e as suas raízes, $r_1, \ldots, r_k$.
2. Determine os coeficientes α_i, para $i = 1, 2, \cdots, k$, através de um sistema de k equações lineares da forma

   $$\alpha_1 {r_1}^n + \alpha_2 {r_2}^n + \cdots + \alpha_k {r_k}^n = f_n$$

   para k valores de f_n distintos que calculou separadamente, por exemplo, para $n = 0, 1, \ldots, k-1$.

Exercício 17 *Aplique o método descrito à solução geral da sequência de Fibonacci (ver página 21). Confirme no final 2 valores da solução obtida.*
RESPOSTA (A PREENCHER PELO ALUNO!):

- *A equação geral é...*
- *A equação característica é...*
- *As raízes da equação característica são...*
- *A solução geral é da forma...*
- *A solução é...*

1.3.8 Equação característica homogénea, caso geral: existência de raízes múltiplas

Teorema 6 *Se m é a multiplicidade de uma raíz r da equação característica, as seguintes sucessões, bem como todas as suas combinações lineares, são soluções da equação geral*

$$r^n,\ nr^n,\ n^2r^n, \cdots,\ n^{m-1}r^n$$

Mais geralmente se as raízes da equação característica forem $r_1, \ldots, r_p$ de multiplicidades respectivamente $m_1, \ldots, m_p$, a solução geral é uma qualquer combinação linear da forma

$$\sum_{i=1}^{i=p} \sum_{j=0}^{m_i-1} \alpha_{ij} n^j {r_i}^n$$

Não é difícil mostrar-se que uma combinação linear da forma indicada é sempre solução da equação geral. Também é verdade o recíproco: qualquer solução da equação geral é da forma indicada!.

Exercício 18 *Resolva a recorrência $t_n = 2t_{n-1} - t_{n-2}$, sendo*

$$\begin{cases} t_0 = 0 \\ t_1 = 1 \end{cases}$$

1.3.9 Método da equação característica não homogénea

Vamos apresentar casos particulares em que se conhece a solução.

Suponhamos que a equação geral da recorrência tem a forma

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = b^n p(n)$$

onde

- b é uma constante
- $p(n)$ é um polinómio em n; seja d o seu grau

Teorema 7 *As soluções da recorrência com a equação geral indicada podem obter-se a partir das*

raízes da equação

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b)^{d+1} = 0$$

usando o método que foi explicado para o caso das equações homogéneas.

Exercício 19 *Determine a forma geral da solução da recorrência*

$$\begin{cases} t_0 = 2 \\ t_n = 2t_{n-1} + 3^n \quad (n \geq 1) \end{cases}$$

Resposta (a preencher pelo aluno!):

- *A equação geral é...*
- *A equação característica é...*
- *Portanto $b = \cdots$, $p(n) = \cdots$, $d = \cdots$.*
- *A solução geral é da forma...*
- *A solução da recorrência é...*

Uma generalização do Teorema 7, equação característica não homogénea.

Teorema 8 *Se a equação geral da recorrência tem a forma*

$$a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \cdots + b_m^n p_m(n)$$

onde os b_i são constantes e os $p_i(n)$ são polinómios em n de grau d_i, então as soluções da recorrência correspondente podem ser obtidas a partir das raízes da equação

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b_1)^{d_1+1}(r-b_2)^{d_2+1}\cdots(r-b_m)^{d_m+1} = 0$$

usando o método que foi explicado para o caso das equações homogéneas.

Exercício 20 *Determine a forma geral da solução da recorrência*

$$\begin{cases} t_0 = 2 \\ t_n = 2t_{n-1} + n + 2^n \quad (n \geq 1) \end{cases}$$

Nota. Será fornecido o enunciado do Teorema 8 nas provas em que isso for necessário.

1.4 Um exemplo de análise: tempo médio do "quicksort"

Supõe-se que o leitor conhece o algoritmo de ordenação "quicksort" que se esquematiza de seguida[1]

```
// ordena a secção v[a..b] do vector v[ ]
void quicksort(v,a,b)
  if a>=b  return
  else
    m = split(v,a,b)
    quicksort(v,a,m-1)
    quicksort(v,m+1,b)
```

O efeito da execução de `m=split(v,a,b)` é alterar o vector `v[]`, definindo um índice `m` tal que todos os elementos da secção do vector `v[a..m-1]` são menores ou iguais a `v[m]` e todos os elementos da secção do vector `v[m+1..b]` são maiores que `v[m]`.

Seja $n = \mathtt{b} - \mathtt{a} + 1$ o número de elementos a ordenar. Na execução de `m=split(v,a,b)` há $n-1$ comparações envolvendo elementos do vector.

Pretendemos mostrar que o número de comparações $c(n)$ é de ordem $O(n \log n)$; mais precisamente que é sempre $c(n) \leq kn \log n$ para uma constante k apropriada.

Admitimos que os elementos do vector são todos distintos e que todas as $n!$ permutações de elementos são igualmente prováveis. Assim, o valor de `m` dado por `m=split(v,a,b)` tem a mesma probabilidade se ser `a`, de ser `a` + 1,... e de ser `b`; essa probabilidade é $1/n$ (relembra-se que $n = \mathtt{b} - \mathtt{a} + 1$).

Obtemos a recorrência que determina o valor médio do número de comparações

$$\begin{cases} c(n) = 0 & \text{para } n \leq 1 \\ c(n) = \sum_{i=1}^{n} \frac{1}{n}(n - 1 + c(i-1) + c(n-i)) & \text{para } n \geq 2 \end{cases}$$

Note-se que para cada $i \in \{1, 2, \ldots, n\}$ o valor $c(i)$ ocorre 2 vezes na soma anterior; podemos então escrever a recorrência da forma seguinte

$$\begin{cases} c(n) = 0 & \text{para } n \leq 1 \\ c(n) = \frac{2}{n} \sum_{i=1}^{n-1}(n - 1 + c(i)) & \text{para } n \geq 2 \end{cases}$$

Pré-teorema *A solução $c(n)$ da recorrência é majorada por $kn \log n$ em que o valor de k é encontrado na demonstração.*

[1] Este algoritmo será analisado em mais detalhe durante este curso.

Caso base, $n = 1$: temos (da recorrência) $c(1) = 0 \leq k(1 \log 1)$ para qualquer $k > 0$; analogamente se trata o caso $n = 0$.

Passo de indução: suponhamos que $c(i) \leq ki \log i$ para todo o i com $0 \leq i \leq n-1$. Vamos mostrar que $c(n) \leq k(n \log n)$.

$$\begin{aligned} c(n) &= n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} c(i) \\ &\leq n - 1 + \frac{2k}{n} \sum_{i=2}^{n-1} (i \log i) \\ &\leq n - 1 + \frac{2k}{n} \int_{i=2}^{n} (i \log i) di \\ &= n - 1 + \frac{2k(n^2 \log n/2 - n^2/4 - 2ln2 + 1)}{n} \\ &= n - 1 + kn \log n - kn/2 + \alpha \end{aligned}$$

onde $\alpha = 2k(-2 \ln 2 + 1)/n$ tem valor negativo. O teorema fica então demonstrado se for $n - 1 \leq kn/2$, pois fica então $k(n \log n)$. Em particular podemos tomar $k = 2$ e enunciar a versão final do "pré-teorema" anterior.

Teorema 9 *O número médio de comparações efectuado pelo "quicksort" não excede* $2n \log n$.

Vemos pois que o número de comparações efectuadas pelo "quicksort" é, no caso médio, majorado por $2n \log n$. Admitimos que o leitor conhece o comportamento no pior caso do "quicksort" e as situações em que esse comportamento se verifica. O número de comparações efectuado é quadrático. Para mais pormenores ver por exemplo o livro de Cormen, Leiserson, Rivest e Stein.

Capítulo 2

Tempo de execução dos algoritmos – complementos

2.1 Tempo de execução

Os recursos mais utilizados para medir a eficiência de um algoritmo A são o tempo de execução e o espaço gasto. Tanto um como outro são função dos dados que representamos por x

$$x \rightarrow \boxed{\text{A}} \rightarrow y$$

Destes 2 recursos consideraremos apenas o tempo de execução que representamos por $t(x)$. Em geral estamos interessados em representar o tempo de execução como função do comprimento (número de bits numa determinada codificação) dos dados, $n = |x|$. Mas pode haver dados do mesmo comprimento que originem tempos de execução muito diferentes, veja-se por exemplo o algoritmo "quick sort"; por outras palavras, t não é função de n, é função de x. O que se faz em geral é tomar uma das seguintes opções

- Considerar o maior dos tempos correspondentes a dados de comprimento n, $T(n) = \max_{|x|=n} t(x)$. A esta medida chama-se tempo no pior caso ("worst case time").
- Considerar o tempo médio correspondente a todos os dados de comprimento n, $\overline{T}(n) = E(t(x))$ supondo-se que todos os dados de comprimento n têm a mesma probabilidade de ocorrência[1]. A esta medida chama-se tempo médio ("average case time").

[1] Na verdade há considerar apenas as sequências de n bits que sejam codificações possíveis das instâncias do problema; a maioria das sequências de bits não faz normalmente sentido como codificação.

Nota. Não havendo ambiguidade, representaremos também por t o tempo no pior caso e o tempo médio.

Por outro lado, não estamos em geral interessados nos valores exactos dos tempos de execução, mas preferimos ignorar factores multiplicativos constantes e comparar os tempos através de *ordens de grandeza*. Este aspecto da questão é tratado, por exemplo nas referências em [1, 2, **?**]. O aluno deve estar familiarizado com as definições e propriedades das ordens de grandeza $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, e $o(\cdot)$. Em muitos casos práticos vai-se pretender apenas obter um limite assimptótico do tempo de execução e ignorar as constantes multiplicativas, isto é, pretende-se estabelecer algo da forma $t(n) \in O(f(n))$ onde $f(n)$ é tão pequeno quanto possível.

2.2 Sobre os modelos de computação

Outra questão que se coloca é "como vamos medir o tempo?". Não será medido certamente em segundos ou micro-segundos mas em termos do número de transições de um modelo de computação apropriado. Há diversas hipóteses; referimos as mais importantes

- *Modelo exacto.* Usamos a máquina de Turing (TM) como modelo de computação e defimos o tempo de uma computação como o número de transições que ocorrem durante a computação.

- *Modelo uniforme.* Usamos um computador de registos e representamos os programas numa linguagem de alto nível ou numa linguagem do tipo "assembler". Para obter o tempo de execução contamos 1 por cada instrução ou teste executado no programa. Dado que, por exemplo, as instruções aritméticas (adição, multiplicação,...) são efectuadas em tempo majorado por uma constante, dito de outra maneira, o seu tempo é $O(1)$, assume-se esse mesmo facto ao medir o tempo de computação.

- *Modelo dos circuitos.* Trata-se de um modelo em que o comprimento dos dados é fixo. Cada circuito "resolve" apenas os problemas cujos dados têm um determinado comprimento. Por outro lado, em vez de um algoritmo, existe um circuito constituído por componentes e ligações. Os componentes (ou circuitos elementares) funcionam em paralelo, o que permite muitas vezes tempos de execução muito mais rápidos. Trata-se de um modelo não-uniforme[2] Falaremos mais do modelo dos circuitos em (6).

O modelo uniforme é muitas vezes bastante conveniente porque está mais próximo das computações usuais, mas está basicamente errado! Não podemos supor que as operações elementares são exe-

[2]O conceito de "uniforme" é usado com 2 sentidos: a *uniformidade* no "modelo uniforme" refere-se à independência do tempo de execução relativamente ao comprimento dos dados, enquanto a *não uniformidade* dos circuitos refere-se ao facto de não existir em princípio uma descrição finita dos circuitos de uma família.

cutadas emtempo $O(1)$, como bem sabem os programadores (por exemplo na área de criptografia) que lidam com inteiros muito grandes. Assim, os algoritmos utilizados usualmente para a adição e a multiplicação têm ordem de grandeza que não é $O(1)$, mas sim $\Theta(\log n)$ e $\Theta(\log^2 n)$ respectivamente, onde n é o maior dos operandos envolvidos. O seguinte exemplo ilustra os graves erros em que se incorre quando se utiliza o modelo uniforme. Considere-se o seguinte algoritmo

```
dado n, inteiro positivo
f(n):
  t=2
  for i=1 to n:
    t = t*t
  return t
```

Usando o modelo uniforme concluimos que o tempo de execução é da forma $a + bn$ onde a e b são constantes. Todavia, se levarmos em consideração o número de bits do resultado e o facto de o processamento de cada um desses bits corresponder a pelo menos um passo da computação, vemos que tal é impossível e verificamos que o tempo real de execução é exponencial no tamanho dos dados.

Exercício 21 *Qual é o valor de* `f(n)` *(expressão matemática fechada)?*

Exercício 22 *Quantos bits tem a representação desse valor na base 2? Que pode afirmar sobre o número de dígitos da representação desse valor numa base arbitrária?*

2.2.1 O modelo externo dos dados

Diversos algoritmos, como por exemplo muitos algoritmos de ordenação e pesquisa, obtêm informação sobre os dados de uma única forma: os dados são comparados entre si. Nestes algoritmos podemos pensar que não existe leitura de dados, mas apenas os resultados das comparações referidas. Um diagrama esquemático destes algoritmos é o seguinte

Por exemplo, um algoritmo de ordenação baseado no modelo externo dos dados, comporta-se exactamente da mesma maneira – isto é, as sucessivas configurações internas são as mesmas – a ordenar `[4,1,8,5,2]` ou `[41,1,8488,55,20]`.

2.2.2 Monotonia de $t(n)$

Para facilitar a análise de um algoritmo, assume-se frequentemente que n tem uma forma específica, por exemplo, $n = 2^p$ ou $n = 2^p - 1$ para um determinado inteiro p. Por outro lado, a forma do algoritmo resulta quase sempre o facto de $t(n)$ (tempo no pior caso ou no caso médio) ser uma função monótona no sentido seguinte

$$[n' \geq n] \Rightarrow [t(n') \geq t(n)]$$

Esta monotonia de $t(n)$ facilita a determinação da ordem de grandeza de $t(n)$ no caso geral, isto é, quando n não tem a forma que se admitiu. Vejamos um exemplo. Seja[3] $t(n) = n \log n$ e $n = 2^p$. Então para um comprimento arbitrário n' (que não é necessariamente uma potência de 2), considere-se o inteiro p tal que $n = 2^p \leq n' < 2^{p+1}$, isto é, $n \leq n' < 2n$. Pela monotonia de $t(n) = n \log n$ temos

$$t(n) = n \log n \leq t(n') \leq 2n \log(2n)$$

Donde $t(n') \leq 2n \log(2n) \leq 2n' \log(2n') = 2n'(1 + \log n')$ e temos também $t(n') \geq n \log n \geq (n'/2) \log(n'/2) = (n'/2)(\log n' - 1)$ o que mostra que se verifica $t(n') \in \Theta(n' \log n')$.

Exercício 23 *Considere a seguinte definição de $t(n)$ para $n \geq 2$:*

$$t(n) = c + \sum_{i=1}^{n-1} t(i)$$

sendo c é uma constante e $t(1) = a$, um valor conhecido. Temos

$$t(n+1) - t(n) = c + \left(\sum_{i=1}^{n} t(i) \right) - c - \left(\sum_{i=1}^{n-1} t(i) \right) = t(n)$$

Resulta: $t(n+1) = 2t(n)$, donde temos a solução $t(n) = a2^{n-1}$.

Esta demonstração está errada! Explique porquê e determine a solução correcta.

[3] ln e log representam respectivamente o logaritmo natural e o logaritmo na base 2.

Exercício 24 *Considere a seguinte função para determinar o menor elemento de um vector* `a[1..n]`*:*

```
Dados: vector a[1..n]
Resultado: min{a[i] : i em {1,2,...,n}}
     min(a):
       sort(a)      -- por ordem crescente
       return a[1]
```

Do ponto de vista da eficiência, esta função é (muito) má. Explique porquê.

2.3 Análise amortizada de algoritmos

Na análise amortizada dos algoritmos considera-se uma sequência de operações e divide-se o tempo total pelo número de operações efectuadas, obtendo-se assim um "tempo médio por operação". Mais geralmente, pode haver operações de vários tipos, sendo atribuído a cada uma um seu custo amortizado.

O conceiro de "custo" generaliza o conceito de "tempo"; o custo é uma grandeza não negativa, em geral inteira, com que se pretende medir os gastos de um determinado recurso como o tempo (caso mais habitual) ou o espaço.

Definição 2 *O* custo amortizado *de uma sequência de n operações é o custo total a dividir por n.*

Quando temos uma sequência de operações op_1, op_2, op_n, podemos considerar o custo máximo de uma operação ("pior caso")

$$c_{\max} = \max\{c_i : i = 1, 2, \ldots, n\}$$

onde c_i representa o custo da operação op_i; o custo médio por operação ("caso médio") ou *custo amortizado* é

$$E(c) = \frac{\Sigma_{i=1}^{n} c_i}{n}$$

Mais geralmente podem existir vários tipos de operações numa estrutura de dados (por exemplo, pesquisa, inserção e eliminação num dicionário) e podemos atribuir a cada tipo de operação um custo amortizado.

Não se devem confundir estes custos (máximo e médio) com os tempos no pior caso e no caso médio (ver página 35), em que a média e o máximo são relativas à distribuição probabilística dos dados.

A análise amortizada é muitas vezes mais significativa do que a análise individual; por exemplo, ao efectuar um determinado processamento de dados, estamos muitas vezes mais interessados no custo total (ou no custo médio por operação) do que no custo individual de cada operação. Por exemplo, se no departamento de informática de um banco se processa, nas horas noturnas, uma longa sequência de transações de diversos tipos, não é importante se o processamento de algumas delas demorar muito tempo, desde que o tempo total não ultrapasse um limite pré-estabelecido. *Se estivessemos a utilizar sistematicamente os custos no pior caso, poderíamos chegar a majorantes do custo total muito elevados e pouco significativos.*

Por outro lado, em sistemas de tempo real (como por exemplo em jogos, nos sistemas de controlo automático associados à navegação aérea, etc.) a análise amortizada poderá não ser conveniente, uma vez que há muitas vezes que garantir majorantes do tempo de cada operação. Pelas mesmas razões, e falando agora da análise baseada na distribuição estatística dos dados, nestes casos prefere-se muitas vezes a análise no pior caso (relativamente à distribuição estatística dos dados) à análise no caso médio.

Vamos de seguida considerar em algum detalhe algumas sequências específicas de operações e fazer a sua análise amortizada.

2.3.1 "Stack" com gestão de memória

Consideremos um "stack" implementado como um vector de dimensão pré-definida

```
Implementação: vector v[0...m-1]
Inicialização: sp=0   (sp é o primeiro índice não ocupado de v)
Operações:
   push(x):    v[sp]=x;   sp++;
   pop():      sp--;      return v[sp];
```

Quando se efectua um `push` e o vector está "cheio", isto é, `sp==m`, há que efectuar previamente a seguinte operação

```
Redimensionar(p):    (onde p>m)
   1) obter um novo espaço de memória com p posições consecutivas
   2) copiar v[0..m-1] para as primeiras m posições do novo espaço
   3) fazer com que v designe para o novo espaço
```

Vamos usar o seguinte modelo de custos:

- `push`: custo 1
- `pop`: custo 1
- `redimensionar`: custo `m`; na verdade, a cópia de `m` elementos domina o tempo de execução desta operação.

A sequência de operações que vamos considerar consiste em n `push`'s. Claro que esta sequência vai eventualmente originar operações de `redimensionar`, mas vamos apenas atribuir um custo amortizado às operações de `push` (e não às de `redimensionar`). Vamos ainda supor que inicialmente o vector tem apenas uma célula disponível, isto é, que o valor inicial de `m` é 0.

Uma questão importante que pode afectar o custo amortizado é: que valor de `p` (como função de `n`) se deve escolher na operação de `redimensionar(p)`? Vamos analisar 2 hipóteses.

1) `p = n + 1`, o vector cresce de 1

Esta é uma má escolha. Vejamos:

	1	2	3		n
op:	push(.)	push(.)	push(.)	...	push(.)
custo:	1	2	3		n

Por exemplo, o custo do último `push` consiste em $n-1$ (operação `redimensionar` que envolve a cópia de $n-1$ valores) mais 1 (custo do `push`). Assim, o custo amortizado é

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

isto é, cada `push` tem um custo superior a $n/2$, não sendo portanto $O(1)$!

2) `p = 2n`, o vector cresce para o dobro

Esta opção é muito melhor, pois vai permitir um custo $O(1)$. Sendo $2^k < n \leq 2^{k+1}$, o custo para os redimensionamentos é

$$1+2+4+8+\ldots+2^k = 2^{k+1}-1$$

a que temos de juntar o custo n dos `push`'s. O custo amortizado é

$$\frac{n+2^{k+1}-1}{n} < \frac{n+2^{k+1}}{n} \leq \frac{3n}{n} = 3$$

pois $2^{k+1} = 2 \times 2^k < 2n$. Assim, o custo amortizado de cada `push` é inferior a 3. Este custo amortizado esconde o facto de alguns `push`'s terem um custo muito superior, $\Omega(n)$.

Exercício 25 *Mostre que com uma análise mais cuidada, podemos usar para o mesmo problema o seguintes* custos amortizados*:*

- `push`*: custo 3*
- `pop`*: custo -1*

Pretende-se mostrar que com este modelo a função potencial continua a nunca ser negativa.

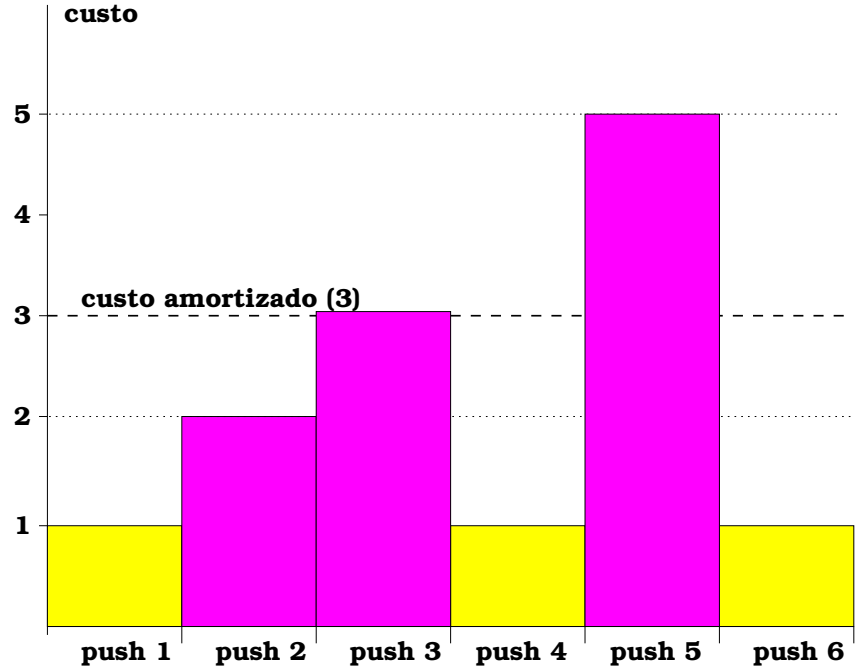
2.3.2 A função potencial

Vamos considerar de novo a sequência de operações anterior (n operações de `push`) com a opção 2 (redimensionamento que duplica o tamanho do vector). Um outro modo de analisar a o custo amortizado desta sequência de operações é o seguinte

- Por cada `push` gastamos 3 unidades (3 euros, por exemplo); dessas 3 unidades, 1 é gasta com o `push` e 2 vão para um saco de poupanças ("mealheiro"); em cada instante, o valor existente no mealheiro é o valor da função potencial.

- Quando há que fazer um redimensionamento, vai-se ao mealheiro buscar o custo respectivo.

Ao conteúdo do mealheiro chamamos "potencial", ver mais à frente a definição formal. Na figura seguinte representa-se o custo de cada operação numa sequência de 6 `push`'s.



Exercício 26 *Represente na figura anterior a função potencial (após cada operação de* `push`*).*

Como podemos concluir que no mealheiro existe sempre o dinheiro suficiente para "pagar" os redimensionamentos? Ou, por outras palavras, como mostrar que a função potencial nunca tem um valor negativo?

Resposta: usando a indução matemática no número de operações. O caso base é trivial. Quando há um redimensionamento de 2^i para 2^{i+1}, é porque $n = 2^i + 1$. Então já houve $n = 2^i$ `push`'s

cada um dos quais contribuiu com 2 unidades para o saco, isto é, para o incremento da função potencial. Mas, após o último redimensionamento (antes do actual, quando o saco continha $n/2$), houve mais $n/2$ contribuições de 2 unidades para o saco, ou seja, antes do `push` actual (que vai obrigar a um redimensionamento de custo n), o saco contém pelo menos $2 \times n/2 = n$ unidades de dinheiro. Note-se que estamos a supor que o último redimensionamento foi possível, isto é, estamos a utilizar a hipótese indutiva.

Definição 3 *A função* potencial $\Phi(s)$ *é uma função do estado s do sistema que satisfaz*

1. $\Phi(0) = 0$*, começa com o valor 0.*

2. $\forall i$, $\Phi(i) \geq 0$*, a função potencial nunca é negativa.*

A função potencial $\Phi(s)$ é usada para a análise amortizada dos algoritmos.

Sejam: a uma quantidade positiva (a vai ser o custo amortizado), c_i o custo da i-gésima operação e $\Phi(0) = 0$, $\Phi(1)$, $\Phi(2)\ldots$ os sucessivos valores da função potencial e suponhamos que o "pagamento" da i-ésima operação é efectuado da seguinte forma

– O valor a é adicionado a $\Phi(i-1)$

– De $\Phi(i-i)$ retira-se o custo da operação corrente, c_i

de modo que $\Phi(i) = \Phi(i-i) + a - c_i$. Então, e supondo que Φ nunca é negativa, o valor a é um majorante do custo amortizado. Na verdade, somando as igualdades $\Phi(i) = \Phi(i-i) + a - c_i$ para $i = 1, 2, \ldots, n$ temos

$$\begin{aligned} \Sigma_{i=1}^{n}\Phi(i) &= \Sigma_{i=1}^{n-1}\Phi(i) + na - \Sigma_{i=1}^{n}c_i \\ na &= \Sigma_{i=1}^{n}c_i + \Phi(n) \\ a &\geq \frac{\Sigma_{i=1}^{n}c_i}{n} \end{aligned}$$

Por outras palavras, a é aceitável como custo amortizado, pois garante-se já "ter pago" em cada instante n todos os custos reais até esse momento.

Em resumo, dada uma constante positiva a (custo amortizado) e os custos das operações c_1, $c_2,\ldots$, uma função potencial $\Phi(i)$ é uma função não negativa definida pela recorrência $\Phi(0) = 0$, $\Phi(i) = \Phi(i-i) + a - c_i$; ao verificarmos que, para os valores dados de a, c_1, $c_2,\ldots$, o valor de $\Phi(i)$ nunca é negativo, estamos a mostrar que a é um (majorante do) custo amortizado.

2.3.3 Outro exemplo, um contador binário

Consideremos um contador representado na base 2 e uma sequência de n operações de incrementar (de 1 unidade). O nosso modelo de custo (de cada incremento) vai ser o número de bits do contador que se modificam. Este custo representa razoavelmente o custo real e leva em particular em conta a propagação dos "carries". Por exemplo, se o contador `010111` é incrementado, fica `011000`, sendo o custo 4.

Vamos mostrar que, embora o custo de algumas operações possa ser elevado, o custo amortizado não ultrapassa 2.

Os primeiros 8 incrementos:

bit 3	bit 2	bit 1	bit 0	custo	potencial
0	0	0	0		0
0	0	0	1	1	1
0	0	1	0	2	1
0	0	1	1	1	2
0	1	0	0	3	1
0	1	0	1	1	2
0	1	1	0	2	2
0	1	1	1	1	3
1	0	0	0	4	1

Análise com base numa função potencial

Vamos definir a seguinte função potencial

$$\Phi(x) = \text{número de 1's em } x$$

(ver figura anterior, "os primeiros 8 incrementos"). É óbvio que esta função satisfaz as condições 1 e 2. da definição. Vamos ver que esta função potencial corresponde ao custo amortizado de 2. Consideremos uma operação arbitrária de incremento,

$$\texttt{xx...x0}\overbrace{\texttt{11...11}}^{m} \rightarrow \texttt{xx...x1}\overbrace{\texttt{00...00}}^{m}$$

onde `x` representa um bit arbitrário e $m \geq 0$ (número de 1's consecutivos no final da representação binária). De uma forma mais compacta, representamos esta transição por $x01^m \rightarrow x10^m$.

Para mostrar que o custo amortizado 2 corresponde exactamente à função de potencial escolhida, analisemos em primeiro lugar a variação da função potencial numa transição $x01^m \rightarrow x10^m$. Representando por $n_1(x)$ o número de 1's de x, temos

– Potencial antes da operação: $n_1(x) + m$

– Potencial depois da operação: $n_1(x) + 1$

Por outras palavras, o número de 1's varia de $1 - m$ (diminui se $m = 0$, aumenta se $m \geq 1$).

Vamos agora ver que este potencial é igual ao valor acumulado (ainda não gasto). Usamos a indução. O caso base (inicial) é trivial. Consideremos novamente a transição $x01^m \rightarrow x10^m$ e suponhamos, pela hipótese indutiva, que o valor acumulado antes da operação é $n_1(x) + m$. De quanto varia o valor acumulado? A contribuição (custo amortizado) é 2 e há $m + 1$ bits que mudam; logo, o valor acumulado varia de $2 - (m + 1) = 1 - m$ (esta variação pode ser positiva, nula ou negativa), o que é exactamente a variação do número total de 1's.

Observação. Após a primeira operação, o número de 1's nunca vai ser nulo, o que equivale a dizer que no fim, o potencial vai ser positivo.

Outra análise

Consideremos n incrementos. Quantas vezes o bit de ordem 0 se modifica? Todas, ou seja n. E o bit de ordem 1? Resposta: $\lfloor n/2 \rfloor$. No total, o número de mudanças de bit, isto é, o custo total, é

$$n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots + 1 \leq n + n/2 + n/4 + \ldots \quad \text{(soma infinita)} \tag{2.1}$$

$$= 2n \tag{2.2}$$

Assim concluímos que o custo amortizado é inferior a 2.

2.3.4 Contador binário com custo exponencial na ordem do bit

Consideremos agora a operação de incremento com outro modelo de custo: modificar um bit de ordem i custa 2^i, exponencial em i. Apesar deste aumento substancial do custo[4], o custo amortizado é apenas de ordem $O(\log n)$.

Exercício 27 *Demonstre esta afirmação, utilizando uma análise análoga à efectuada anteriormente, ver a desigualdade (2.1).*

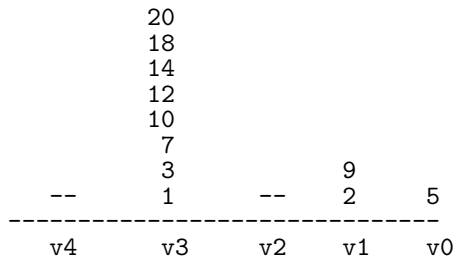
[4]Este modelo pode ser adequado noutras situações como, por exemplo, na análise do custo de um acesso à memória num sistema com uma hierarquia de memórias; além disso, este modelo vai ter uma aplicação na estrutura de dados estudada em seguida.

Aplicação: estrutura de dados com pesquisa e inserção

Vamos definir uma estrutura de informação baseada numa sequência de vectores v_0, v_1, $v_2 \ldots$ em que o vector v_i ou está vazio ou contém 2^i elementos e está ordenado.

Exercício 28 *Mostre que qualquer número de elementos pode ser representado numa estrutura deste tipo.*

Um exemplo desta estrutura



Não se impõe qualquer relação entre os valores contidos nos diferentes vectores.

Exercício 29 *Mostre que a pesquisa de um valor x numa estrutura deste tipo (isto é, x está em algum dos vectores?) pode ser efectuada em tempo $O(\log^2 m)$ onde m é o número total de elementos contidos na estrutura..*

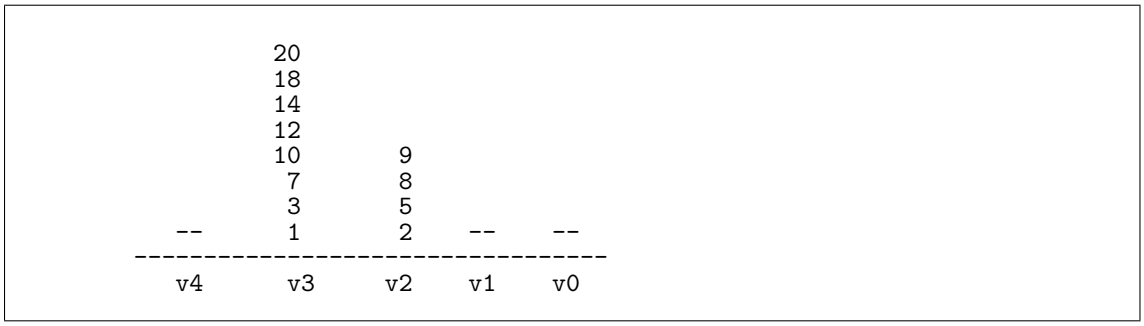
A inserção de um elemento x na estrutura utiliza operações de "merge", conforme se exemplifica para a inserção de 8; seja o vector $w = [x]$

1. Se v_0 está vazio, faça $v_0 = w$, STOP.
2. Se não: faça merge(w,v_0) $\rightarrow w$ (2 elementos). No caso presente fica $w = [5, 8]$. Depois marque v_0 como vazio.
3. Se v_1 está vazio, faça $v_1 = w$, STOP.
4. Se não, faça merge(w,v_1) $\rightarrow w$ (4 elementos). No caso presente fica $w = [2, 5, 8, 9]$. Depois marque v_1 como vazio.
5. Se v_2 está vazio, faça $v_2 = w$, STOP (caso presente).

6. Se não...

7. ...

A estrutura após a inserção de 8:



Modelo de custos:

– Criação do vector inicial: custo 1

– "Merge" de 2 vectores de tamanho k (cada): custo $2k$

Exercício 30 *(i) Qual o custo da inserção de 8 exemplificada atrás? (ii) Mostre que se o valor inserido ficar no vector v_i (inicialmente vazia), então o custo é $\Theta(2^i)$.*

Exercício 31 *Mostre que o custo amortizado da inserção de n valores numa estrutura inicialmente vazia é $O(\log(n))$. Sugestão: utilize o exercício anterior.*

Capítulo 3

Sobre o esquema "Dividir para Conquistar"

3.1 Uma recorrência associada ao esquema "dividir para conquistar"

Suponhamos que uma implementação do esquema genérico "dividir para conquistar" tem a seguinte forma

```
    função f(problema de tamanho n):
      if n=1:
        return <solução imediata>
      else:
(1)     divide-se o problema em b partes de tamanho n/b
        fazem-se a chamadas recursivas
(2)         f(problema de tamanho n/b)
(3)     combinam-se as a soluções
        ... no resultado que é retornado
```

Para simplificar vamos supor que n é potência de b, seja $n = b^m$. Admitindo que a divisão (1) e a combinação (3) são executadas num tempo de ordem $O(n^k)$, temos a seguinte recorrência para a definição de um majorante do tempo de execução

$$t(n) = at(n/b) + cn^k \tag{3.1}$$

Para simplificar supomos que $t(1) = c$. Para estudar a solução desta recorrência, começemos por

considerar o caso $n = b^2$. Temos

$$\begin{aligned} t(n) &= at(n/b) + cn^k && (3.2)\\ &= a(at(n/b^2) + c(n/b)^k) + cn^k && (3.3)\\ &= a^2 t(1) + acb^k + cb^{2k} && (3.4)\\ &= a^2 c + acb^k + cb^{2k} && (3.5) \end{aligned}$$

Note-se que nesta expressão a, b e k são constantes; a variável é m com $n = b^m$, ou seja $m = \log_b n$.

No caso geral obtemos

$$t(n) = c(a^m + a^{m-1}b^k + \cdots + a^0 b^{mk}) = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

Casos a considerar: $a > b^k$, $a = b^k$, $a < b^k$.

Caso 1. $a > b^k$

Trata-se de uma soma geométrica. Sendo $r = b^k/a$ é $r < 1$ e temos

$$t(n) = ca^m \frac{r^{m+1} - 1}{r - 1} < \frac{ca^m}{1 - b^k/a} \in O(a^m)$$

Mas $a^m = a^{\log_b n} = b^{(\log_b a)(\log_b n)} = n^{\log_b a}$.

Caso 2. $a = b^k$

Neste caso os termos da soma são iguais e $t(n) = c(m+1)a^m \in O(ma^m)$. Como $m = \log_b n$, é

$$n^k = b^{(\log_b n)(log_b a)} = a^{\log_b n}$$

donde

$$ma^m = (\log_b n)a^{\log_b n} = (\log_b n)n^k \in O(n^k \log n)$$

Caso 3. $a < b^k$

Procedendo como no primeiro caso e sendo $r = b^k/a$, temos $r > 1$ e

$$t(n) = ca^m \frac{b^{k(m+1)}/a^{m+1} - 1}{b^k/a - 1} \in O(b^{km}) = O(n^k)$$

Em resumo, temos

Teorema 10 *A solução de uma recorrência com a equação geral da forma $t(n) = at(n/b) + cn^k$ onde c e k são inteiros positivos, a e b são inteiros com $a \geq 1$, $b \geq 2$ tem a seguinte ordem de*

grandeza

$$\begin{cases} t(n) \in O(n^{\log_b a}) & \text{se } a > b^k \\ t(n) \in O(n^k \log n) & \text{se } a = b^k \\ t(n) \in O(n^k) & \text{se } a < b^k \end{cases}$$

O resultado mais forte que resulta de substituir em cima a ordem de grandeza $O(\cdot)$ por $\Theta(\cdot)$ é também válido.

Utilizaremos este resultado diversas vezes neste curso.

Exercício 32 *Considere os algoritmos "merge sort" e "pesquisa binária"; mostre como o resultado anterior pode ser utilizado em cada um desses algoritmos para encontrar a correspondente ordem de grandeza do tempo de execução.*

3.2 Multiplicar mais rapidamente...

Em primeiro lugar vamos considerar a multiplicação de 2 inteiros de n bits, isto é, de grandeza compreendida entre 2^{n-1} e $2^n - 1$, e depois consideraremos a multiplicação de 2 matrizes quadradas de dimensão $n \times n$. Os resultados serão surpreendentes.

3.2.1 Multiplicação de inteiros

Consideremos 2 inteiros de n bits, ou um pouco mais geralmente, 2 inteiros x e y em que o maior deles tem n bits, $n = \max\{|x|, |x|\}$. Um primeiro método se obter o produto xy consiste em somar y parcelas todas iguais a x,

```
9482 * 2596:     9482   +
                 9482       (2596 parcelas!)
                 ....
                 9482
                ------
              24615272
```

ou, em algoritmo

```
Algoritmo de multiplicação (somas sucessivas)

            prod1(x,y):
              s=0
              for i=1 to y:
                s=s+ x
              return s
```

Este algoritmo é terrivelmente ineficiente, sendo exponencial no tamanho dos dados n: o número de adições é $\Omega(2^n)$ (sendo $|x| = |y| = n$) e a cada adição corresponde um tempo de $\Theta(n)$.

O método "escolar", que todos conhecemos de cor, é muito mais eficiente

```
 9482 * 2596:     9482   x
                  2596
               --------
                  56892
                 85338
                47410
               18964
              ---------
               24615272
```

ou, em algoritmo

```
Algoritmo escolar da multiplicação de inteiros.
Representação de b na base 10: b = b[n-1]b[n-2]...b[1]b[0]

            prod2(x,y):
              s=0
              shift=0
              for i=0 to n-1:
                s=s + shift(x*y[i],i) -- shift i posições para a esquerda
              return s
```

(é claro que na base 2, a operação `a*b[i]` é trivial) Este algoritmo é muito mais eficiente, na realidade de ordem $\Theta(n^2)$ uma vez que para cada `i` são efectuadas n adições (cada uma em tempo $\Theta(n)$) mais um "shift" que não altera a ordem de grandeza.

Será este o método mais eficiente? Vamos mostrar que não. Em 1962 o russo Anatoli Karatsuba descobriu um método mais eficiente (com uma ordem de grandeza inferior a n^2), baseado na ideia de "dividir para conquistar". Consideremos os inteiros x e y com (não mais de) $2n$ bits

$$\begin{cases} x = a \times 2^n + b \\ y = c \times 2^n + d \end{cases}$$

Os primeiros n bits de x "estão em" a e os últimos n em b; e analogamente para y. Efectuemos o produto xy:

$$xy = ac2^{2n} + bc2^n + ad2^n + bd \tag{3.6}$$

Na realidade, esta igualdade não nos vai ajudar muito em termos de eficiência: para multiplicar 2 números de $2n$ bits temos de efectuar 4 multiplicações de inteiros de n bits, isto é, vamos ter um tempo de execução definido pela recorrência

$$\begin{cases} t(1) = k \\ t(2n) = 4t(n) + cn \end{cases}$$

onde k e c são constantes. O termo cn representa um majorante do tempo de execução das 3 adições, dos "shifts" bem como do tempo de procesamento dos "carries". A expressão anterior corresponderá a uma multiplicação do tipo

Exercício 33 *Mostre que a solução da recorrência anterior é de ordem* $\Theta(n^2)$.

Então, relativamente à ordem de grandeza do algoritmo "escolar", nada se ganhou com esta aplicação do "dividir para conquistar"! A melhoria da eficiência assimptótica resulta da observação crucial seguinte: é possível reescrever a igualdade (3.6) usando apenas 3 multiplicações de inteiros de n (ou $n+1$) bits

$$xy = ac(2^{2n} - 2^n) + (a+b)(c+d)2^n + bd(1-2^n) \tag{3.7}$$

Pode-se, efectuando as multiplicações de (3.7), mostrar a equivalência a (3.7). E agora temos apenas 3 multiplicações, para além de diversas operações ("shifts" e adições) implementáveis em tempo linear. Temos então um tempo de execução definido pela recorrência seguinte (onde c é uma nova constante):

$$\begin{cases} t(1) = k \\ t(2n) = 3t(n) + cn \end{cases}$$

Exercício 34 *Mostre que a solução da recorrência anterior é de ordem* $\Theta(n^{\log_2 3}) = \Theta(n^{1.5849\ldots})$.

Este método é assintóticamente mais rápido, embora o "overhead" – representado pelas constantes que as ordens de grandeza escondem – seja maior. Só vale a pena aplicar este método quando os inteiros envolvidos são muito grandes.

Será este o método mais rápido que existe para a multiplicação de inteiros? Não. O algoritmo FFT ("Fast Fourier Transform") foi usado por Karp para mostrar que é possível multiplicar 2 inteiros de n bits em tempo $\Theta(n \log^2(n))$ e mais tarde (em 1971) Schönhage e Strassen descobriram um algoritmo de eficiência $\Theta(n \log n \log(\log n))$ que é, em termos de ordem de grandeza, o algoritmo de multiplicar mais rápido que se conhece.

3.2.2 Multiplicação de matrizes

A história aqui vai ser semelhante à da secção anterior. Consideramos a multiplicação de 2 matrizes quadradas de dimensão $n \times n$ e vamos utilizar o modelo uniforme, supondo que cada adição ou multiplicação elementar é efectuada em tempo $O(1)$; trata-se de uma hipótese realista se os inteiros contidos nas matrizes não são muito grandes (cabem numa "palavras" do computador).

Tal como na secção anterior vamos ver que uma aplicação elementar do esquema "dividir para conquistar" não leva a qualquer redução da ordem de grandeza, sendo necessário usar observações não triviais para haver sucesso. Esses "feitos" são: na secção anterior (multiplicação de inteiros grandes) conseguir escrever uma expressão com apenas 3 (em vez de 4) multiplicações de inteiros e na presente secção a redução de 8 para 7 do número de multiplicações matriciais numa determinada expressão.

Exercício 35 *Mostre que no algoritmo usual de multiplicação de 2 matrizes quadradas de dimensão* $n \times n$

- *O número de multiplicações elementares é* n^3
- *O número de adições elementares é* $n^2(n-1)$
- *Conclua que, usando o modelo uniforme, a complexidade do algoritmo usual é* $\Theta(n^3)$.

Se dividirmos cada uma das matrizes de dimensão $2n \times 2n$ em 4 sub-matrizes, temos que efectuar o produto

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix} \tag{3.8}$$

As operações que levam à matriz resultado

$$R = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix}$$

são pois as seguintes

$$\begin{cases} R_{11} &= AE+BG \\ R_{12} &= AF+BH \\ R_{21} &= CE+DG \\ R_{22} &= CF+DH \end{cases} \tag{3.9}$$

As adições são efectuadas em tempo $\Theta(n^2)$, havendo 8 chamadas recursivas; assim o tempo de execução é definido por

$$\begin{cases} t(1) = k \\ t(2n) = 8t(n) + cn^2 \end{cases}$$

Exercício 36 *Mostre que a solução $t(n)$ desta recorrência satisfaz $t(n) \in \Theta(n^3)$.*

Não se ganhou nada em termos de tempo de execução. Nem sequer na realidade se mudou basicamente o algoritmo.

Notemos que num algoritmo recursivo implementado segundo um esquema baseado em (3.8) apenas vai haver recursão na multiplicação de matrizes, sendo as adições efectuadas imediatamente; a ordem de grandeza do tempo das adições é $\Theta(n^2)$. Strassen mostrou que o produto 3.8 pode ser efectuado com apenas 7 multiplicações (em vez de 8). As operações que levam à matriz resultado R são as seguintes (confirme!) onde os M_i são matrizes temporárias

$$\begin{cases} M_1 &= (A+D)(E+H) \\ M_2 &= D(G-E) \\ M_3 &= (B-D)(G+H) \\ M_4 &= (A+B)H \\ M_5 &= (C+D)E \\ M_6 &= A(F-H) \\ M_7 &= (C-A)(E+F) \\ R_{11} &= M_1+M_2+M_3-M_4 \\ R_{12} &= M_4+M_6 \\ R_{21} &= M_2+M_5 \\ R_{22} &= M_1-M_5+M_6+M_7 \end{cases}$$

Compare-se com (3.9) e repare-se que agora há apenas 7 multiplicações de matrizes de $n \times n$. O tempo de execução é pois da ordem de grandeza definida pela seguinte recorrência

$$\begin{cases} t(1) = 1 \\ t(2n) = 7t(n) + cn^2 \end{cases}$$

Exercício 37 *Mostre que a solução desta recorrência é de ordem* $\Theta(n^{\log 7})$.

O algoritmo de Strassen só é vantajoso na prática para o cálculo do produto de matrizes de grande dimensão.

O algoritmo de Strassen é o mais rápido? Mais uma vez, não. Tem havido diversas melhorias em termos de ordens de grandeza, sendo actualmente o algoritmo de Coppersmith e Winograd o mais rápido.

Há ainda um aspecto que é importante mencionar. Nas implementações concretas, quando as chamadas recursivas chegam a uma dimensão n suficientemente pequena recorre-se ao algoritmo clássico de ordem n^3, pois para pequenos valores de n este é muito mais rápido. Por outro lado, os algoritmos pós-Strassen têm "constantes escondidas" tão grandes que se tornam impraticáveis; os valores de n para os quais seriam mais rápidos são gigantescos. Em conclusão: as ordens de grandeza não nos "contam a história toda". Por exemplo, se os tempos de execução dos algoritmos A e B são respectivamente n^2 e $150n^{2.8}$, embora B seja assintóticamente mais rápido que A, isso só acontece para valores de n superiores a $150^5 \approx 76\,000\,000\,000$!

Capítulo 4

Algoritmos aleatorizados e classes de complexidade

Neste capítulo vamos ver que, mesmo para problemas que não têm nada a ver com probabilidades ou estatística, como por exemplo o problema da ordenação de um vector, o acesso a números aleatórios pode ser benéfico, no sentido em que se obtêm algoritmos mais eficientes (no caso médio), independentemente da distribuição dos dados. Ilustraremos as vantagens da aleatorização dos algoritmos com 2 exemplos: um algoritmo de coloração (página 57), um algoritmo que testa uma dada matriz é o produto de outras 2 matrizes (página 59) e ainda o algoritmo "quick sort" (página 60). Em seguida (página 73), classificaremos os algoritmos aleatorizados e definiremos classes de complexidade baseadas em máquinas de Turing que têm acesso a uma sequência de bits aleatórios. O modo de obter inteiros verdadeiramente aleatórios ou então "pseudo-aleatórios" e o modo como os segundos podem aproximar ou não os primeiros, são questões (interessantes!) que não são tratadas nesta secção.

Em secções posteriores apresentamos um teste aleatorizado de primalidade devido a Rabin e Miller e discutiremos em termos mais gerais os algoritmos aleatorizados e as classes de complexidade aleatorizadas.

4.1 Um problema de coloração

Vamos agora considerar o seguinte problema:

Problema "colorir". É dado um conjunto S com n elementos e uma família A_1, A_2,..., A_m de sub-conjuntos distintos de S, cada uma contendo exactamente r elementos. Pretende-se atribuir a cada elemento de S uma de 2 cores (azul e vermelho, por exemplo) por forma a que em cada sub-conjunto A_i exista pelo menos um elemento com uma das cores e um elemento com a outra cor.

Note-se que cada elemento de S pode pertencer a mais que uma das famílias A_i (pode até pertencer a todas). Por exemplo, S poderia ser um conjunto de alunos, já inscritos em disciplinas A_i (de forma que cada turma tenha exactamente r alunos inscritos) e pretende-se entregar a cada aluno um computador ou uma impressora (mas não as 2 coisas!) por forma que em cada disciplina exista pelo menos um aluno com um computador e pelo menos um aluno com uma impressora.

Nota importante. Para que o algoritmo aleatorizado que vamos definir funcione convenientemente, vamos supor que $m \leq 2^{r-2}$.

A primeira ideia para resolver este problema leva geralmente a um algoritmo bastante complicado. Mas acontece que o seguinte algoritmo aleatorizado resolve o problema em pelo menos 50% dos casos:

```
Algoritmo aleatorizado básico para o problema da coloração
-----------------------------------------------------------
Para cada elemento de x de S:
  atribuia-se a x, aleatoriamente e uniformemente (prob=1/2),
    a cor azul ou a cor vermelha
Verifique-se se a coloração está correcta
Se estiver: retorna a solução, senão retorna "não definido"
```

Nada mais simples! É claro que este algoritmo pode errar (ou melhor, retornar `"não definido"`), mas vamos ver que a probabilidade de isso acontecer não ultrapassa 1/2. Vamos demonstrar esse facto.

Mas vejamos primeiro um exemplo: seja $n = 9$, $m = 4$ e $r = 4$ (verifica-se pois a condição $m \leq 2^{r-2}$) e os conjuntos

$$A_1 = \{1, 2, 3, 4\},\ A_2 = \{3, 5, 8, 9\},\ A_3 = \{1, 5, 6, 7\},\ A_4 = \{2, 3, 6, 9\}$$

Atirei a moeda ao ar e obtive a coloração: 1, 2, 3, 4, 5, 6, 7, 8, 9 (para quem está a ler a preto e branco: 2, 3, 4, 6, 7 e 8 são vermelhos e os restantes azuis), É fácil verificar (verifique!) que tivemos sorte, pois obtivemos uma solução.

Para qualquer i com $1 \leq i \leq m$ a probabilidade de todos os elementos do conjunto A_i terem a mesma cor é 2^{-r+1} (porquê?). E a probabilidade de pelo menos um A_i não ser válido, isto é,

de todos os seus elementos terem a mesma cor, não excede $m2^{-r+1}$ (porquê?). Assim, e usando a hipótese $m \leq 2^{r-2}$, concluimos que a probabilidade da coloração "à sorte" não estar correcta não excede

$$m2^{-r+1} \leq 2^{r-2}2^{-r+1} = \frac{1}{2}$$

E o que é que obtivemos? *Um algoritmo aleatorizado que, independentemente dos dados, acerta em pelo menos 50% das vezes.*

Claro que 50% é pouco, mas vamos ver na Secção 4.4 como tornear o problema.

4.2 O produto de duas matrizes iguala uma terceira?

Suponhamos que conhecemos 3 matrizes quadradas A, B e C, todas de dimensões $n \times n$, e que pretendemos saber se $AB = C$. O método mais imediato consiste em multiplicar A por B e ver se o resultado é igual a C. Se usarmos o algoritmo clássico de multiplicação de matrizes, a complexidade deste método é de ordem $\Theta(n^3)$. Mesmo utilizando os métodos assimptoticamente mais rápidos, não conseguimos melhor que $\Theta(n^{2.373\ldots})$.

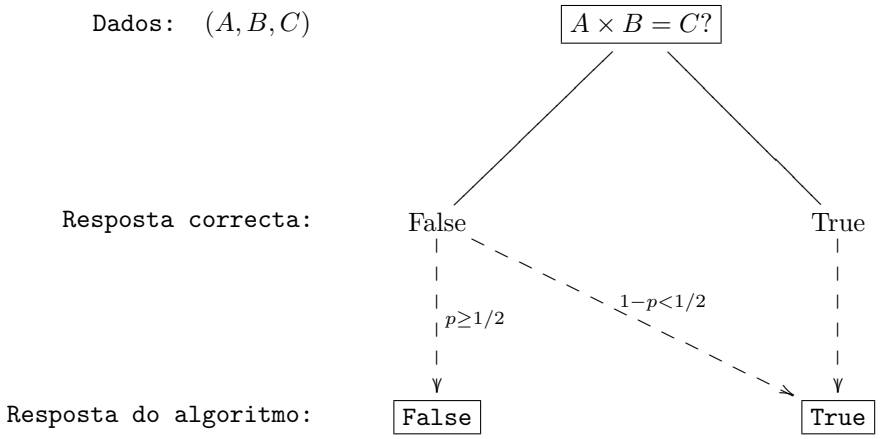
Vamos apresentar um algoritmo aleatorizado (que pode errar) que é muito simples e rápido.

```
Algoritmo que testa se A*B=C
----------------------------

  Dados: Matrizes quadradas, n*n, A, B, C
  Resultado: True se A*B=C, False caso contrário

1    - Gera-se um vector coluna r com n elementos em que cada elemento
       é 0 ou 1 com probabilidade 1/2 (de forma independente)
2    - Calcula-se x=Br (vector coluna) e depois y=Ax (vector coluna)
3    - Calcula-se z=Cr (vector coluna)
4    - Se y=z: retorna True
       senão:  retorna False
```

A ordem de grandeza do tempo de execução é $\Theta(n^2)$, sendo dominada pelas produtos matriciais (linhas 2 e 3). Independentemente dos dados, temos

$$\begin{cases} \text{se } AB = C & \rightarrow \quad \text{resposta } \texttt{True} \\ \text{se } AB \neq C & \rightarrow \quad \text{resposta } \texttt{False} \text{ com probabilidade } \geq 1/2 \end{cases}$$

No diagrama seguinte representamos por p a probabilidade da resposta do algoritmo ser `False`.



Notas.

– Este algoritmo funciona em qualquer corpo $(A, +, \times)$.

– Embora a probabilidade de o algoritmo errar (isto é, responder `True` quando $A \times B \neq C$) possa ser relativamente grande, a repetição da execução do algoritmo para os mesmos dados permite reduzir esse erro a valores exponencialmente pequenos; mais especificamente, com k execuções do algoritmo, a probabilidade de erro não excede 2^{-k}. Ver a Secção 4.4.

4.3 O "quick sort"

Começaremos por analisar o caso médio e o pior caso do algoritmo clássico do "quick sort" e depois uma versão aleatorizada desse algoritmo em que o pivot é escolhido (de cada vez que o algoritmo corre) de forma aleatória.

4.3.1 Esquema básico do "quick sort"

Vamos supor que todos os elementos do vector a ordenar são distintos; se tal não acontecer, é fácil alterar o algoritmo e a sua análise por forma a poder haver elementos iguais.

```
Quick sort clássico
-------------------
Dados: um vector x[a..b] a ordenar

1    Se o número de elementos do vector (b-a+1) é 0 ou 1:
       HALT
2    Escolhe um "pivot", isto é um dos x[p], com a <= p <= b
     Seja piv=x[p]
3    (Split) particiona-se o vector x[a..b] em 3 partes:
       x[a..m-1]: com elementos x[i]<piv
       x[m]:      com o valor piv, x[m]=piv
       x[m+1..b]: com elementos x[i]>piv
4    Recursivamente aplica-se o algoritmo a x[a..m-1]
5    Recursivamente aplica-se o algoritmo a x[m+1..b]
```

Não se trata ainda de um um algoritmo[1], uma vez que a parte de particionamento do vector (linha 3) não está especificada. Mas é suficiente para os nossos propósitos.

4.3.2 Análise no pior caso do "quick sort" clássico.

Vamos supor que escolhemos como pivot o primeiro elemento, `piv=x[1]`. Se fosse escolhido qualquer outro elemento de índice fixo como pivot, as conclusões seriam semelhantes.

Seja `n=b-a+1` o número de elemento do vector. Se o vector já estiver ordenado, `x[a]`$<$`x[a+1]`$<\ldots<$`x[b]`, após a primeira partição fica o lado esquerdo do vector (elementos menores que `piv`) vazio e o lado direiro com `n-1` elementos. A partição demora tempo $\Theta(n)$ (efectuam-se $n-1$ comparações). Em seguida efectuam-se chamadas recursivas a vectores de tamanhos `n-1`, `n-2`,..., `2`. O tempo total (e o número de comparações é pois de ordem $\Theta(n^2)$. O número exacto de comparações é $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$. Obviamente há outros casos com este mau comportamento (em termos da ordem de grandeza), como o do vector `[1,8,2,7,3,6,4,5]`; o que não há é casos com ordem de grandeza pior que $\Theta(n^2)$, uma vez que a linha 2 (escolha do pivot) pode ser efectuada no máximo `n-1` vezes (de cada vez que é efectuada há necessariamente menos um elemento, o pivot, que não faz parte dos sub-vectores a ordenar) e de cada vez na linha 3 fazem-se no máximo `n-1` comparações.

Em conclusão, no pior caso, o "quick sort" é de ordem $\Theta(n^2)$.

4.3.3 Análise do tempo médio do "quick sort" clássico

Assumimos aqui que o aluno tem conhecimentos básicos de probabilidades. Por simplicidade, continuamos a supor que todos os elementos do vector são distintos.

[1] Em certas linguagens de "alto nível" é possível escrever o "quick sort" de forma curta e elegante. Tais linguagens não existiam ainda quando este algoritmo foi inventado por C. A. R. Hoare em 1960! Num dos apêndices (pág. 170) pode ver-se implementações do "quick sort" em Haskell, Prolog e Python.

Ao particionar o vector com `n` elementos fazem-se `n - 1` comparações, uma vez que o pivot é comparado com cada um dos outros elementos para definir o sub-vector da esquerda e o sub-vector da direita. Dependendo do valor do pivot, podem acontecer os seguintes casos

Num. de elementos à esquerda	Num. de elementos à direita
0	$n-1$
1	$n-2$
2	$n-3$
...	...
$n-2$	1
$n-1$	0

Como o pivot (o primeiro elemento do vector) tem igual probabilidade de ser o menor, o segundo menor,..., o maior, qualquer uma das partições da tabela anterior tem a mesma probabilidade, $1/n$. Então a recorrência que define o *valor esperado* do número de comparações é a seguinte

$$\begin{cases} t(0) = 0 \\ t(1) = 0 \\ t(n) = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(t(i) + t(n-i-1)) \end{cases}$$

Note-se que $t(0) = 0$ e que cada um dos restantes $t(i)$ ocorre duplicado. Por exemplo, para $n = 8$, aparece o somatório

$$\begin{aligned} t(4) &= 3 + [(t(0) + t(4)) + (t(1) + t(3)) + (t(2) + t(2)) + (t(3) + t(1)) + (t(4) + t(0))] \\ &= 3 + 2[(t(1) + t(2) + t(3) + t(4))] \end{aligned}$$

Assim a equação geral da recorrência pode escrever-se

$$t(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} t(i)$$

A solução desta recorrência vai seguir os 2 passos tradicionais: inspiração e demonstração.

- *Inspiração.* Vamos admitir que em quase todos os casos o particionamento é tal que a parte esquerda e a parte direita são sensivelmente iguais. Se fossem sempre exactamente iguais[2], a altura máxima da árvore associada às chamadas recursivas não inferior a $\log n$. Por outro lado, *em cada nível de recursão* o número total de comparações não pode exceder n (aliás é sempre inferior a n). Assim, esperamos que, eventualmente, o número total de comparações

[2] A altura mínima ocorre quando n for da forma $n = 2^p - 1$ e os pivots são sempre a mediana do sub-vectores a ordenar.

seja limitado por $cn \log n$ para alguma constante $c > 0$. A seguir vamos demonstrá-lo.

– *Demonstração.*

Pretendemos mostrar que o valor $t(n)$ definido pela recorrência satisfaz $t(n) \leq cn \ln n$ para uma constante c a determinar; é mais conveniente usar o logaritmo natural e, dado o facto de termos mais tarde que determinar a constante c por forma a que a proposição seja verdadeira, a base do logaritmo é indiferente. Os casos base são triviais. Para o passo de indução temos

$$\begin{aligned} t(n) &= (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} t(i) && (4.1)\\ &\leq (n-1) + \frac{2c}{n}\sum_{i=1}^{n-1} i \ln i && (4.2)\\ &\leq (n-1) + \frac{2c}{n}\int_1^n x \ln x dx && (4.3)\\ &\leq (n-1) + \frac{2c}{n}\left(\frac{1}{2}n^2 \ln n - \frac{n^2}{4} + \frac{1}{4}\right) && (4.4)\\ &\leq cn \ln n && (4.5) \end{aligned}$$

desde que $c \geq 2$. Os passos que foram usados são

- Da linha (4.1) para a linha (4.2): hipótese indutiva.
- Da linha (4.2) para a linha (4.3): se $f(x)$ é positivo e crescente em $[1, n]$ então a soma $f(1)+f(2)+\ldots+f(n-1)$ é majorada pelo integral $\int_1^n f(x)$ (basta usar a interpretação dos integrais definidos como áreas).
- Da linha (4.3) para a linha (4.4): cálculo do integral definido.
- Da linha (4.4) para a linha (4.5): propriedade fácil de verificar.

Assim, podemos enunciar o seguinte resultado

Teorema 11 *Supondo-se que todas as permutações do vector inicial são igualmente prováveis, o valor esperado do número de comparações efectuadas no algoritmo clássico do "quick sort" não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução deordem* $O(n \log n)$.

4.3.4 O "quick sort" aleatorizado

Consideremos agora uma pequena alteração ao "quick sort" clássico: o índice correspondente ao elemento pivot é escolhido aleatóriamente de cada vez que o algoritmo corre e de forma (estatis-

ticamente) independente da distribuição dos dados[3]

```
Quick sort aleatorizado
-----------------------
Dados: um vector x[a..b] a ordenar

1    Se o número de elementos do vector (b-a+1) é 0 ou 1:
       HALT
2    É escolhido aleatoriamente e uniformemente (com igual
     probabilidade) um índice p de {a,a+1,...,b}.
     Seja piv=x[p]
3    Particiona-se o vector x[a..b] em 3 partes:
       x[a..m-1]: com elementos x[i]<piv
       x[m]:      com o valor piv, x[m]=piv
       x[m+1..b]: com elementos x[i]>piv
4    Recursivamente aplica-se o algoritmo a x[a..m-1]
5    Recursivamente aplica-se o algoritmo a x[m+1..b]
```

A alteração é mínima, mas os efeitos na complexidade são grandes.

Consideremos um qualquer vector dado `v[]` fixo. Toda a análise que se fez na secção anterior (páginas 61 a 63) continua válida uma vez que, devido à linha 2 do algoritmo, qualquer índice tem a mesma probabilidade de ser escolhido como índice do pivot. Temos assim o resultado seguinte:

Teorema 12 *Para qualquer vector dado fixo, o valor esperado do número de comparações efectuadas pelo algoritmo "quick sort" aleatorizado não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução de ordem* $O(n \log n)$*.*

Obviamente, e como corolário deste teorema, se o vector dado não for fixo mas obedecer a uma distribuição probabilística qualquer, continua a verificar-se o resultado anterior.

Corolário 1 *Para qualquer distribuição probabilística do vector dado, o valor esperado do número de comparações efectuadas pelo algoritmo "quick sort" aleatorizado não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução deordem* $O(n \log n)$*.*

O algoritmo clássico e o algoritmo aleatorizado são quase iguais; no último o modelo de computação (o computador) é probabilístico pois tem acesso a uma fonte de números aleatórios. Em termos de variáveis aleatórias, a variável aleatória Y associada ao resultado y é função de 2 variáveis aleatórias independentes:

- A variável aleatória X associada ao vector dado x
- A variável aleatória R associada ao modelo de computação que deve ser necessariamente independente de Y e uniforme.

[3]Vamos continuar a supor que todos os elementos do vector a ordenar são distintos.

E o que é que obtivemos com a aleatorização do algoritmo?

Qualquer que seja a distribuição probabilística associada aos dados (incluindo mesmo casos extremos como o caso em que a probabilidade é 0 para todos os vectores possíveis excepto para um) o tempo médio de execução do algoritmo é de ordem $O(n \log n)$. Por outras palavras, em termos do tempo médio, não existem más distribuições dos dados.

Comparemos com o que se passa com o algoritmo clássico:

Embora o tempo médio de execução seja de ordem $\Theta(n \log n)$, qualquer que seja o pivot escolhido (fixo) existem vectores (e portanto existem distribuições probabilísticas dos dados) para os quais o tempo médio de execução do algoritmo é de ordem $O(n^2)$.

4.4 Técnica de redução da probabilidade de erro

Vamos apresentar uma técnica de reduzir a probabilidade de erro de um algoritmo aleatorizado; essa técnica consiste essencialmente na repetição do algoritmo. Embora este método seja bastante geral, será explicado com o problema da secção anterior (problema de coloração).

Suponhamos que corremos o algoritmo 2 vezes. Temos 2 colorações. Qual a probabilidade de pelo menos uma estar correcta? A probabilidade de estarem as 2 erradas é não superior a[4]

$$\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$$

Assim, a probabilidade de pelo menos uma estar correcta é pelo menos $1 - 1/4 = 3/4$. Mais geralmente, se correrrmos o algoritmo k vezes, a probabilidade de pelo menos numa das vezes o algoritmo produzir uma solução (correcta) é não inferior a $1 - 1/2^k$.

Podemos então escolher uma das 2 hipóteses seguintes

- Correr o algoritmo, digamos um máximo de 500 vezes. A probabilidade de não ser obtida uma solução correcta é não superior a $2^{-500} \approx 3 \times 10^{-151}$, um valor ridiculamente pequeno, igual a 0 para todos os efeitos práticos[5]. No algoritmo seguinte, o algoritmo aleatorizado básico de coloração (página 58) é executado no máximo k vezes, onde k é um parâmetro fixo. A probabilidade de ser retornada uma solução é pelo menos $1 - 2^{-k}$.

[4] Uma vez que a probabilidade da conjunção de 2 acontecimentos independentes é o produto das respectivas probabilidades.

[5] "Quem acha que 10^{100} não é infinito deve ter perdido o seu juizo", John von Neumann.

```
Algoritmo aleatorizado para o problema da coloração
com probabilidade exponencialmente pequena de não ser retornada uma
solução.
-------------------------------------------------------------------------
Seja COL o algoritmo aleatorizado básico para o problema da coloração
-------------------------------------------------------------------------

  for i=1 to n:
      Execute COL(S,A1,A2,...Am)
      se é retornada uma solução s:
         return s
  return "não definido"
```

– Correr o algoritmo até ser produzida uma solução. O número de vezes que é necessário correr o algoritmo para ser obtida uma solução é uma variável aleatória com um valor médio muito pequeno

$$\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + \ldots = 2$$

4.5 Outro algoritmo aleatorizado: o algoritmo de Rabin-Miller

O problema de determinar se um inteiro n é primo, o chamado problema da primalidade, é um problema básico na Teoria dos Números e tem aplicações muito importantes na área da criptografia. Assim por exemplo, os 2 factos seguintes

– A existência de algoritmos eficientes para o problema da primalidade

– A densidade relativamente elevada dos primos[6]

permitem a geração de números primos grandes com razoável eficiência, tal como se exige no protocolo RSA.

4.5.1 Ineficiência dos algoritmos elementares de primalidade

Os algoritmos elementares de primalidade são exponenciais, ver o exercício seguinte.

Exercício 38 *Mostre que os seguintes algoritmos são exponenciais (em $|n| \approx \log n$)*

a) Testar se cada um dos inteiros $2, 3, \ldots, \lfloor n/2 \rfloor$ *é divisor de* n.

[6] Teorema...

a) Testar se cada um dos inteiros 2, 3, ..., $\lfloor\sqrt{n}\rfloor$ *é divisor de* n.

Nota. *Mesmo se nos limitarmos a testar como possíveis divisores os primos não superiores a* $\lfloor\sqrt{n}\rfloor$*, o algoritmo continua a não ser polinomial.*

4.5.2 Existem algoritmos polinomiais para a primalidade

Apenas recentemente[7] se demostrou que o problema da primalidade pertence a classe **P**, isto é, que pode ser resolvido por um algoritmo determinístico em tempo polinomial, no pior caso. Contudo, o algoritmo proposto neste trabalho não se usa na prática pois, embora seja polinomial, envolve constantes multiplicativas muito grandes e é de ordem elevada.

4.5.3 Testemunhos rarefeitos da não primalidade

Um número que não é primo diz-se *composto*. Os números compostos têm testemunhos. Designamos por *testemunho* de um determinado facto uma informação (por exemplo, uma palavra ou um inteiro) que permite verificar a validade desse facto em tempo polinomial. Já encontrámos esta noção quando estudámos a classe de complexidade **NP**. Um possível testemunho de um inteiro n ser composto é um divisor não trivial de n. Baseado nesses testemunhos, podemos definir o seguinte algoritmo aleatorizado para determinar se um número é primo

```
Algoritmo aleatorizado para o problema da primalidade
Dado: n
Resultado:
   se n é primo: resposta SIM
   se n é composto: a resposta pode ser NÃO (correcta)
                    mas também pode ser SIM (errada)

(*) m = inteiro aleatório uniforme em {2,3,...,n-1}
   se m divide n:
      return NÃO
   senão:
      return SIM
```

O problema deste algoritmo é que a probabilidade de erro pode ser exponencialmente próxima de 1, não podendo ser limitada superiormente por uma constante inferior a 1; caso contrário, seria possível, por repetição das experiências (correr o algoritmo várias vezes com o mesmo n) reduzir a probabilidade de erro a um valor exponencialmente baixo; quando dizemos "exponencialmente", estamos a falar em função do número de vezes que o algoritmo corre, ver a Secção 4.4. Porque é

[7] Manindra Agrawal, Neeraj Kayal and Nitin Saxena, *PRIMES is in P* foi publicado um algoritmo polinomial (determinístico) para o teste da primalidade.

que não é possível limitar superiomente o erro por uma constante menor que 1? Consideremos por exemplo, a infinidade de inteiros da forma p^2 em que p é primo. De entre o número exponencial de inteiros que podem ser gerados na linha `(*)` do algoritmo anterior, apenas um, nomeadamente p, é testemunha de o número ser composto; assim, em determinados casos a probabilidade de erro é demasiado grande (da forma $1 - \Theta(2^{-n})$).

O que necessitamos é de outros testemunhos da não primalidade que sejam sempre frequentes, isto é, tais que (dando um possível exemplo) qualquer que seja n não primo, mais de metade dos inteiros compreendidos entre 1 e $n-1$ sejam testemunhos. Se conseguirmos esta densidade de testemunhos, o seguinte algoritmo

```
  Forma do algoritmo aleatorizado que se pretende
  Dado: n
  Resultado:
     se n é primo: resposta SIM
     se n é composto: resposta incorrecta com probabilidade <= 1/2

(*) m = inteiro aleatório uniforme (possível testemunha) ...
    se m é testemunha:
       return NÃO
    senão:
       return SIM
```

4.5.4 Testemunhos frequentes da não primalidade

Para definirmos testemunhos da não primalidade que sejam frequentes, qualquer que seja o inteiro composto, vamo-nos socorrer do chamado "pequeno teorema de Fermat".

Teorema 13 *Seja p um inteiro primo. Então, qualquer que seja o inteiro a com $1 \leq a \leq p-1$ é $a^{p-1} = 1 \pmod p$.*

Vejamos o exemplo $p = 5$; temos $1^4 = 1$, $2^4 = 16$, $3^4 = 81$, $4^4 = 256$. O resto da divisão por 5 de todos estes inteiros é 1.

Dem. Consideremos os sucessivos produtos (em $\mathbb{Z}_p$)

$$a,\ 2a,\ \ldots,\ (p-1)a$$

Se 2 destes produtos fossem iguais, seja $\alpha a = \beta a \pmod p$ e teríamos $\alpha = \beta \pmod p$, o que é falso uma vez que o maior divisor comum entre p e qualquer dos coeficientes 1, 2, $p-1$ é 1. Assim, aqueles $p-1$ produtos são congruentes, numa qualquer ordem, com[8] 1, 2,..., $p-1$. Então,

[8] No exemplo que antecede esta prova (em que $a = 4$ e $p = 5$), os sucessivos valores de ka módulo p são sucessivamente 4, 3, 2 e 1.

multiplicando as $p-1$ congruencias temos

$$\begin{aligned} a \times 2a \times \ldots \times (p-1)a &= 1 \times 2 \times \ldots \times (p-1) \pmod p \\ a^{p-1}(p-1)! &= (p-1)! \pmod p \\ a^{p-1} &= 1 \pmod p \end{aligned}$$

A passagem da primeira equação para a segunda resulta da observação feita acima, enquanto a terceira equação se obtém da segunda, dividindo (em $\mathbb{Z}_p$) por $(p-1)!$; note-se (verifique), que $p-1$ não é nulo. □

Assim, temos um novo algoritmo aleatorizado, baseado no novo testemunho de não-primalidade de n: um inteiro a, compreendido entre 2 e $n-1$ tal que $a^{n-1} \neq 1 \pmod p$.

```
Algoritmo aleatorizado para o problema da primalidade
Dado: n
Resultado:
   se n é primo: resposta SIM
   se n é composto: resposta incorrecta com probabilidade ???
                    (a determinar)

(*)  m = inteiro aleatório uniforme em {2,3,...,n-1}
(+)  x = m^(n-1)  (mod n)

     se x=1:
        return PRIMO
     senão:
        return COMPOSTO
```

Para este algoritmo ser eficiente, há que efectuar a potência (linha `(+)`) de forma eficiente. Isso exige

1. Sempre que, no cálculo da potência, se efectua um produto deve-se tomar o resto da divisão por n. Isso evita que haja inteiros de comprimento descomunal e torna razoável o modelo uniforme (uma vez tomando o comprimento máximo da representação de n como parâmetro).

2. O cálculo da potência deve ser feito de modo eficiente. Não basta, por exemplo, efectuar $n-1$ multiplicações por m, o que seria exponencial. Um algoritmo apropriado para a potenciação (escrito em C, para variar) pode ver-se no Apêndice da página 171.

Agora põe-se o problema: será que, sempre que um inteiro n é composto, a frequência destes testemunhos, isto é o quociente

$$\frac{\#\{m : m^{n-1} \pmod n \neq 1\}}{n-2}$$

é, por exemplo, maior ou igual a 0.5 (ou outra constante positiva)?

Fazendo um teste computacional, procurámos inteiros compostosnão superiores a 1000 tais que a frequência dos testemunhos é inferior a 0.5. Quantos encontrámos? Um, 561. Encontramos realmente testemunhos frequentes dos números compostos que funcionam,... quase sempre. O problema é o "quase". Os primeiros inteiros compostos com frequência inferior a 0.5 que encontrámos são

Inteiro	Frequência
561	0.428
1105	0.304
1729	0.250
2465	0.273
2821	0.234
6601	0, 200

Estes inteiros têm um nome, os números de Carmichael, ou pseudo-primos de Fermat. Eles arruinaram o nosso algoritmo aleatórizado de primalidade! Mais informação sobre estes inteiros pode encontrar-se em diversos livros de Teoria dos Números e, por exemplo, em
`http://en.wikipedia.org/wiki/Carmichael_number`.

Felizmente, existe um outro teste que completa este, por forma a que a frequência dos testemunhos seja pelo menos 0.5 (na realidade, pelo menos 0.75).

Teorema 14 *Seja p um primo ímpar, $p-1 = 2^t u$ com u ímpar e seja $1 \leq a \leq p-1$. Então verifica-se uma das proposições seguintes*

a) $a^u = 1 \pmod p$

b) Existe um inteiro i com $1 \leq i \leq t-1$ tal que $a^{2^i u} = -1 \pmod p$.

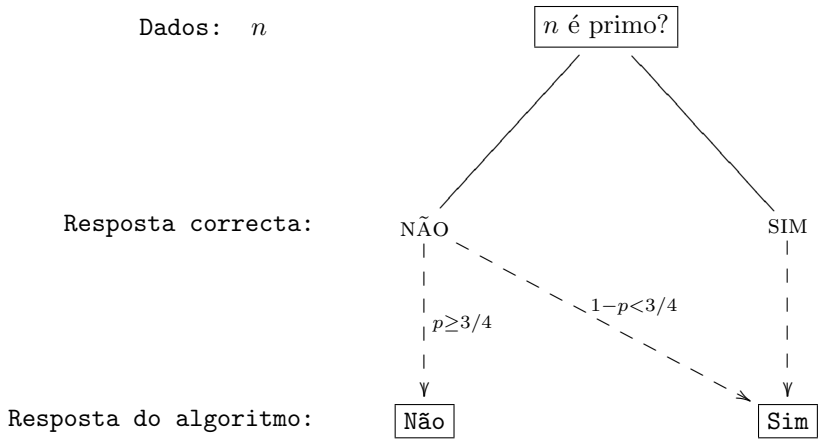
Teorema 15 *Seja n um inteiro composto ímpar, $n-1 = 2^t u$ com u ímpar.*
Se um inteiro $a \in \{2, 3, \ldots, n-1\}$ verificar

$$[a^u \neq 1 \pmod n] \wedge [a^{2^i u} \neq -1 \pmod n \text{ para todo o } i \in \{1, 2, \ldots, t\}]$$

dizemos que a é uma <u>testemunha</u> de n ser composto.

Teorema 16 *Para n ímpar e para um inteiro $a \in_u \{2, 3, \ldots, n-1\}$ (escolha aleatória uniforme), a probabilidade de a ser uma testemunha da não-primalidade de n é pelo menos $3/4$.*

Este Teorema permite finalmente encontrar um algoritmo aleatorizado que se comporta da forma seguinte



O algoritmo é o seguinte:

```
Algoritmo aleatorizado de Rabin-Miller para o problema da primalidade
Dado: n, inteiro ímpar
Resultado:
   se n é primo: resposta SIM
   se n é composto: resposta incorrecta com probabilidade <= 3/4
                    (a determinar)

   Escreva n-1 na forma 2^t * u
   a = inteiro aleatório uniforme em {2,3,...,n-1}

   x = u^(n-1)  (mod n)
   se x diferente de 1:
      return COMPOSTO
   senão:
      calcule (eficientemente):
        a^u, a^(2u), a^(3u),..., a^((t-1)u)
      se algum destes valores for congruente com -1 (mod. n):
          return PRIMO    (provavelmente)
      senão:
          return COMPOSTO
```

Exercício 39 *Mostre que o algoritmo anterior pode ser implementado em tempo polinomial (em termos de* $\log n$*).*

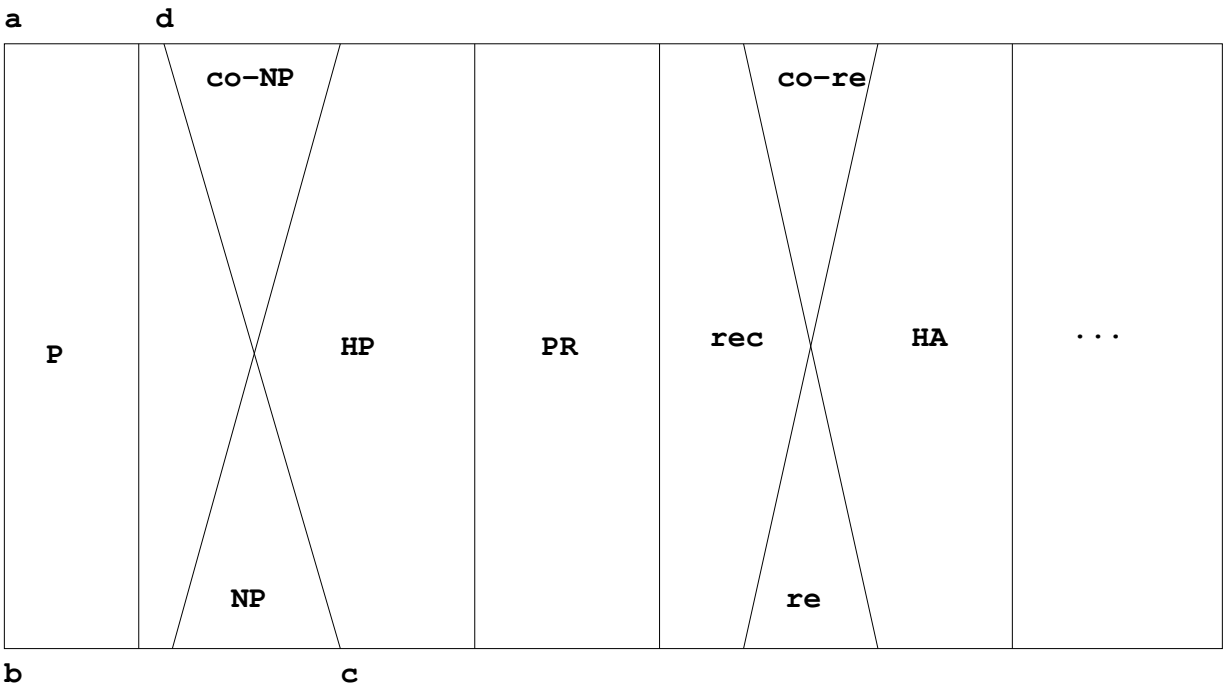
Exercício 40 *Suponha que pretende saber se um inteiro dado é primo ou não com uma probabilidade de erro inferior a* $0.000\,000\,001$*. Quantas vezes deve correr o algoritmo anterior para garantir esse majorante do erro? Se obtiver o resultado "*`COMPOSTO`*", que pode concluir? E se obtiver o resultado "*`PRIMO`*"?*

4.6 Computação aleatorizada: classes de complexidade

4.6.1 Panorama geral das classes de complexidade

O diagrama seguinte contém um sumário das classes de complexidade e de computabilidade dos predicados[9] Deve notar-se o seguinte

- Cada classe inclui todas as classes que estão para a esquerda; por exemplo, a classe **NP** corresponde à área do trapézio `a-b-c-d`.
- De forma implícita, o diagrama anterior define também as diferenças entre classes.
- A classe dos predicados decidíveis é a classe `rec`, a qual inclui evidentemente todos as classes à esquerda.



Legenda:

- **P** : Classe dos predicados decidíveis em tempo polinomial.
- **NP** : Classe dos predicados decidíveis em tempo polinomial por uma máquina de Turing não determinística.
- **HP** : Hierarquia polinomial, $\bigcup_{i\geq 0}(\Sigma_i^p \cup \Pi_i^p)$
- `PR`: Classe dos predicados definíveis por recursão primitiva.

[9] Quando falamos em "predicados"podemos em alternativa falar em "linguagens"; é basicamente a mesma coisa; ao predicado $p(x)$ corresponde a linguagem $L = \{x : p(x)\}$ e à linguagem L corresponde o predicado $x \in L$.

– `rec`: Classe dos predicados recursivos.

– `re`: Classe dos predicados recursivivamente enumeráveis.

– `co-re`: Classe complementar de `re`.

– **HA** : Hierarquia aritmética, $\bigcup_{i\geq 0}(\Sigma_i \cup \Pi_i)$

Este diagrama está ainda muito incompleto. Indicamos dois exemplos desse facto: (i) a classe **P** pode ser sub-dividida em sub-classes que traduzem a dificuldade de solução dos predicados polinomiais. (ii) A classe dos problemas completos em NP, ou "NP-completos", também não está indicada (pertence a **NP****P**).

4.6.2 Classes de complexidade aleatorizadas

Vamos agora definir as classes aleatorizadas mais importantes. Vamos utilizar um esquema de definição semelhante ao usado em [7]; esse esquema inclui também as classes **P** e **NP**.

Representamos por x os dados do problema e por y uma variável aleatória que é a testemunha do algoritmo aleatorizado ou do problema **NP**. Seja $n = |x|$ e $l(n)$ uma função de n majorada por um polinómio, sendo $|y| = l(n)$. Quando escrevemos $y \in_u \{0,1\}^{l(n)}$ pretendemos dizer que y tem a distribuição uniforme (isto é, a mesma probabilidade, igual a $1/2^{l(n)}$) em $\{0,1\}^{l(n)}$.

Definição.

– **P** : Classe das linguagens L cuja função característica $f\colon \{0,1\}^n \to \{0,1\}$ é computável em tempo polinomial.

– **NP** : Classe das linguagens L tais que existe uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$ tal que

$$\begin{cases} x \in L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] > 0 \\ x \notin L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] = 0 \end{cases}$$

– **RP** : Classe das linguagens L para as quais existe uma constante $\varepsilon > 0$ e uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$ tais que

$$\begin{cases} x \in L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] \geq \varepsilon \\ x \notin L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] = 0 \end{cases}$$

– **BPP** : Classe das linguagens L para as quais existem constantes $\varepsilon > 0$ e ε' com $0 \leq \varepsilon < 1/2 < \varepsilon' \leq 1$ e uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$

tais que

$$\begin{cases} x \in L & \Rightarrow \quad \text{prob}_{\,y \in_u \{0,1\}^{l(n)}}[f(x,y)=1] \geq \varepsilon' \\ x \notin L & \Rightarrow \quad \text{prob}_{\,y \in_u \{0,1\}^{l(n)}}[f(x,y)=1] \leq \varepsilon \end{cases}$$

Exercício 41 *Mostre que esta definição da classe* **NP** *coincide com a definição que conhece.*

Exercício 42 *Se conhecer (apenas) o algoritmo aleatorizado de Rabin-Miller para a primalidade, o que pode dizer sobre a classe do problema da primalidade (ou da linguagem correspondente)? E, na realidade, a que classe pertence?*

Exercício 43 *Mostre que*

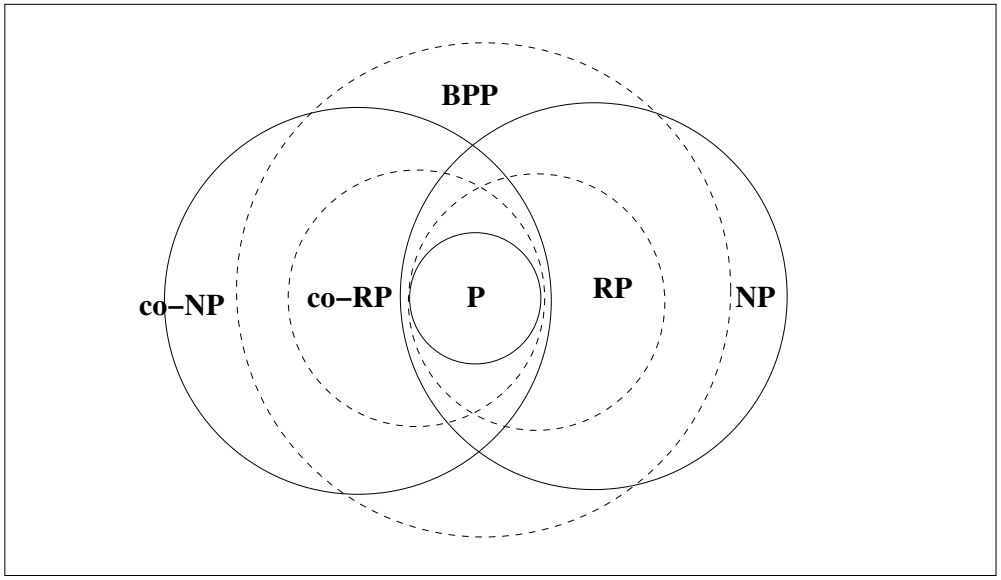
a) **RP**$\subseteq$**NP**, *co-*$\mathbf{RP} \subseteq$*co-*$\mathbf{NP}$.

a) co-$\mathbf{BPP} = \mathbf{BPP}$

Exercício 44 *Mostre que poderíamos ter arbitrado* $\varepsilon = 1/2$ *na definição de* **RP**. *A solução deste problema usa um algoritmo que consiste executar um número fixo um outro algoritmo.*

Exercício 45 *Mostre que poderíamos ter arbitrado* $\varepsilon = 1/3$ *e* $\varepsilon' = 2/3$ *na definição de* **BPP**.

A relação entre estas classes de complexidade esté ilustrada na figura seguinte. Note-se que não há (tanto quanto se sabe) nenhuma relação de inclusão entre **BPP**, **NP** e co-**NP**.


BPP
co−NP
co−RP
P
RP
NP

Capítulo 5

Sobre a ordenação e a selecção

Já conhecemos diversos modos de ordenar uma sequência de valores:

- Métodos elementares de eficiência $O(n^2)$ como a ordenação por selecção do mínimo, o método da bolha ("bubble sort") e a ordenação por inserção.
- Um método com tempo médio de ordem $O(n \log n)$, mas tempo no pior caso de ordem $O(n^2)$: o "quick sort".
- Métodos com tempo médio de ordem $O(n \log n)$, mesmo no pior caso: o "heap sort" e o "merge sort".

Em todos estes métodos a informação sobre o vector é obtida através das comparações que envolvem elementos do vector, (modelo externo dos dados, ver página 37), medindo-se frequentemente a eficiência dos algoritmos através no número de comparações efectuadas. Determinaremos no Capítulo 10 um minorante $c(n)$ para o número de comparações efectuadas, nomeadamente $c(n) \geq \log(n!)$.

Neste capítulo estudaremos alguns métodos que não são baseados em comparações envolvendo elementos do vector, mas sim na utilização directa dos valores a ordenar para indexar um outro vactor; são exemplos a ordenação por contagem e o "bucket sort". Também é possível utilizar partes dos valores a ordenar, como dígitos ou caracteres, para a indexação referida; é isso o que acontece, por exemplo, no "radix sort".

Em seguida voltaremos aos métodos baseados em comparações envolvendo os elementos a comparar e estudaremos métodos eficientes para o um problema aparentado com o da ordenação: dado i e um vector com n elementos, determinar o elemento de ordem i, isto é, o elemento do vector com $i-1$ elementos menores que ele. Este problema inclui como caso particular o

problema da mediana, correspondente ao caso $i = 1 + \lfloor n/2 \rfloor$.

Posteriormente (Capítulo 6) estudaremos as *redes de ordenação* e a *ordenação óptima* – a que efectua o menor número de comparações. Tanto num caso como noutro, estamos a utilizar modelos de computação não uniformes, os circuitos (ou algoritmos não uniformes) de ordenação. Com "não uniforme" pretendemos dizer que a sua estrutura pode depender de n, o número de elementos a ordenar.

Definição 4 *Sejam* `v[]` *e* `w[]` *vectores com* `m` *e* `n` *elementos, respectivamente. Dizemos que a ordenação de* `v[]` *resulta em* `w[]` *(ou que* `v[]` *ordenado é* `w[]`*) se*

- *Todo o elemento de* `v[]` *ocorre em* `w[]` *coma mesma multiplicidade.*
- $\texttt{m} = \texttt{n}$*; como corolário,* `w[]` *"não tem elementos que não estão em* `v[]`*".*
- $\forall i,\ 1 \leq i \leq n-1\colon\ \texttt{w}[i] \leq \texttt{w}[i+1]$.

Quando um programa ordena um vector `v[]`, pode determinar-se para cada i com $1 \leq i \leq n$ qual o índice j em que `v[i]` foi colocado, isto mesmo quando existem elementos iguais no vector.

Definição 5 *Um algoritmo de ordenação diz-se estável se* $\{v[i] = v[j] \wedge i < j\} \Rightarrow i' < j'$*, onde* i' *e* j' *são as posições onde são colocados os elementos que no início estão nas posições de índices* i *e* j*, respectivamente.*

Vamos dar um exemplo muito simples de um algoritmo de ordenação (para um vector com 2 elementos) que não é estável

```
Ordena v[1..2]
if v[1] >= v[2]:
   v[1],v[2] = v[2],v[1]  // troca
```

Se tivermos o vector `v=[5,5]`, os elementos são trocados: temos um caso com $i < j$ mas $i' > j'$.

> **Exercício 46** *Modifique minimamente o algoritmo anterior por forma a que passe a ser estável. ser estável.*

A noção de ordenação estável pode não fazer muito sentido do ponto devista matemático, mas é muito importante do ponto de vista prático. Na prática, os valores que se estão a ordenar são muitas vezes apenas um dos *campos* de *registos* que podem ter muito mais informação associada;

se o vector a ordenar já está ordenado segundo outro campo é natural que se pretenda efectuar umaordenação compatível com a que já existe, isto é estável.

Exercício 47 *Explique através de um exemplo prático o interesse de uma ordenação ser estável.*

5.1 Quando o universo é pequeno: indexação nos valores

Vamos propor um exercício ao leitor. Apenas depois de o resolver, ou de pelo menos o tentar seriamente resolver, deverá continuar com a leitura deste texto.

Exercício 48 *Suponha que os valores a ordenar são inteiros positivos, não muito grandes e sem repetições. Descreva um algoritmo que os ordene sem efectuar qualquer comparação entre os elementos do vector; nenhum operador de relação (<, <=, etc.) deve existir no seu algoritmo.*

Nesta secção supomos que os valores a ordenar pertencem ao conjunto

$$U = \{1, 2, \ldots, u\}$$

onde u é um inteiro não muito grande, no sentido de se poderem usar vectores de tamanho u; $u = 10\,000$ seria aceitável, mas $u = 10^{20}$ não o seria.

5.1.1 Vector sem elementos repetidos

Apresentamos em seguida um algoritmo que ordena o vector `v[]` sem efectuar comparações. A ideia fundamental é:

> *Colocar 1 no vector auxiliar* `c[x]` *(inicialmente com 0 em todos os elementos) sempre que* `x` *é um elemento de* `v[]`*, isto é,* `x==v[i]` *para algum* `i`*.*

```
  Algoritmo de ordenação "por indicação de presença"
    (por vezes chamado de "bucket sort")
  Vector a ordenar: v[1..n], não tem elementos repetidos
  Vector auxiliar:  c[1..u] (toma valores 0 ou 1)

1   for i=1 to u:  c[i] = 0
2   for i=1 to n:
3     c[v[i]] = 1   // faz c[i]=1 se v[i] existe
4   k=0
5   for i=1 to u:   // percorre c
6     if c[i]==1:   // ...colocando (por ordem) em v[] os elementos
7       v[k]=i
8       k=k+1
```

Se não há qualquer comparação, como é obtida informação sobre o vector? Pela operação de indexação, linha 3; cada vez que a instrução `c[v[i]]=1` (linha 3) é efectuada, toda a informação sobre o valor do vector, `v[i]`, é usada para indexar `c[]`.

Análise da eficiência Temos as seguintes contribuições para o tempo de execução do algoritmo anterior

- Linha 1: $O(u)$
- Linhas 2-3: $O(n)$
- Linhas 5-8: $O(u)$
- Linha 4: $O(1)$

O tempo de execuão do algoritmo é $\Theta(n+u)$.

Exercício 49 *Mostre que, sendo n e m parâmetros positivos, $\Theta(n+m) = \Theta(\max\{n,m\})$.*

5.1.2 Comentário: uma representação de conjuntos

O algoritmo descrito sugere a seguinte representação de sub-conjuntos de U: seja $A \subseteq U$; A é representado por um vector booleano `a[1..u]` sendo `a[x]=True` se $x \in A$ e `a[x]=False` caso contrário. No algoritmo anterior os elementos de `v[]` constituem um conjunto que é representado pelo vector `c[1..u]`.

Exercício 50 *Usando a representação descrita para os conjuntos, escreva funções*

- `inters(a,b)` *que retorna a representação da intersecção dos conjuntos representados por* `a` *e* `b`.

– `sim-diff(a,b)` *que retorna a representação da diferença simétrica*[1] *dos conjuntos representados por* `a` *e* `b`.

5.1.3 Vector com elementos repetidos

Uma generalização simples do algoritmo anterior (5.1.1) permite-nos ordenar vectores em que nos elementos podem ocorrer repetições; o que fazemos é usar `c[]`, não para indicar a presença de um elemento x, mas para indicar *quantas vezes* x ocorre em `v[]`.

```
 Algoritmo de ordenação por contagem
 Vector a ordenar: v[1..n]
 Vector auxiliar:  c[1..u] (toma valores inteiros)

1   for i=1 to u:  c[i] = 0
2   // incrementa c[i] se v[i] existe
3   for i=1 to n:  c[v[i]] = c[v[i]]+1
4   k=0
5   for i=1 to u:   // percorre c
6     for j=1 to c[i]:
7       v[k]=i
8       k=k+1
```

Por exemplo

```
                      v                    c
                ----------------------------------------
Início:           [8,5,2,5,5,1]    [0,0,0,0,0,0,0,0,0]
Após linha 3:     [8,5,2,5,5,1]    [0,2,0,0,3,0,0,1,0]
Fim:              [2,2,5,5,5,8]    [0,0,0,0,0,0,0,0,0]
```

Análise da eficiência Temos as seguintes contribuições para o tempo de execução do algoritmo anterior

– Linha `1`: $O(u)$

– Linha `3`: $O(n)$

– Linhas `5-8`: $O(n+u)$; o teste do `for` da linha `6` é efectuado $u+n$ vezes, as linhas `7-8` são executadas exactamente `n` vezes.

– Linha `4`: $O(1)$

O tempo de execução do algoritmo é $\Theta(n+u)$.

Outra versão do algoritmo da contagem Um problema do algoritmo apresentado é perder-se a noção do movimento dos valores associado à operação de ordenar, e consequentemente não fazer sentido averiguar se a ordenação é estável. Na seguinte versão do algoritmo tal não acontece, os valores são de facto movidos (na linha `5`).

```
 Algoritmo de ordenação por contagem
 Vector a ordenar: v[1..n]
 Vector auxiliar:  c[1..u] (toma valores inteiros)
 O resultado fica no vector w[1..n]

1   for i=1 to u:  c[i] = 0
2   for i=1 to n:  c[v[i]] = c[v[i]]+1
3   for i=2 to n:  c[i] = c[i]+c[i-1]
4   for i=n downto 1:
5     w[c[v[i]]] = v[i]
6     c[v[i]] = c[v[i]] -1
```

Embora este algoritmo seja curto, a sua compreensão não é imediata... como acontece muitas vezes que há indexação a mais que 1 nível. Por isso propomos ao leitor o seguinte exercício.

Exercício 51 *Justificar a correcção do algoritmo apresentado. Na sua justificação os seguintes passos podem ser úteis:*

- *Seguir o algoritmo para um pequeno exemplo (com elementos repetidos).*
- *Descrever as variáveis usadas, incluindo o vector* `c[]`*; note-se que o significado da variável* `i` *depende da linha em consideração.*
- *Descrever os efeitos e o conteúdo dos vectores* `v[]`*,* `c[]`*, e* `w[]` *após cada uma das linhas* `1`*,* `2`*,* `3`*,* `4`*,* `5` *e* `6`*.*

Correcção do algoritmo

Vamos resolver o exercício anterior, mostrando que cada elemento `v[i]` é colocado na posição final correcta quando se executa a linha `5` do algoritmo anterior.

Após a execução das linhas `2-3`, `c[x]` contém o número de valores de `v[]` que são menores ou iguais a `x`. Assim, o último valor `x` de `v[]`, seja `v[i]` (isto é, `i` é o índice de `v[]` onde se encontra o último `x`), deve ser colocado na posição `c[x]` (igual a `c[v[i]]`) de `w[]`, isto é, devemos executar a instrução

```
w[c[v[i]]] = v[i]
```

e percorrer `v[]` da direita para a esquerda; *é exactamente isso o que se faz nas linhas* `4-6`. Devido à instrução da linha `6` o próximo valor igual a `x` (se houver) será colocado na posição anterior.

Por exemplo, se `v[]` for `[8,2,5,2,4,5,6,5,7]`, o último (mais à esquerda) `5` deve ser colocado na posição `6`, pois há 6 elementos não superiores a `5` (incluindo este `5`); mas `c[5]` é 6, por construção de `c[]`.

Exercício 52 *Mostre que o algoritmo é estável.*

Nota. A estabilidade do algoritmo resulta também da análise anterior. Devido ao facto de se percorrer `x` da direita para a esquerda e ao decremento da linha `6`, o último valor de `v[]` que é igual a um determinado `x` é colocado após o penúltimo valor de `v[]` que é igual a esse `x`, e assim sucessivamente.

5.1.4 Notas sobre as tentativas de generalização do universo

Os algoritmos que apresentamos na Secção (5.1) estão bastante limitados pelo facto de os elementos a ordenar terem que ser inteiros positivos não muito grandes. Assim houve várias tentativas deos generalizar, entre as quais as seguintes

- Para valores que são inteiros, possivelmente negativos: determinar um inteiro positivo apropriado que somado aos valores os transforme em números positivos.
- Valores reais entre 0 e 1: usar uma variante do "bucket sort" (página 79) com cadeias e indexar em intervalos, ver secção (5.1.5) e Capítulo 9 de [2].
- Se os valores são cadeias de elementos de um determinado alfabeto, usar o método "radix sort" descrito nas secções 5.2.1 e 5.2.2. Este método é aplicável, por exemplo, aos inteiros (representados numa determinada base) e às "strings".

Nota. Com os métodos de "hash" consegue-se usar um tipo muito vasto de valores, mas não é possível normalmente obter a informação disposta por ordem, como seria conveniente para a ordenação dos valores.

5.1.5 Ordenação de reais no intervalo $[0, 1)$

Pretendemos ordenar o vector `v[]` que contém n números reais pertencentes ao intervalo $[0, 1)$. Usamos um vector auxiliar também com n células; cada uma destas células define uma lista ligada de valores, estando inicialmente todas as listas vazias. Cada elemento `v[i]` é colocado no início da lista $\lfloor n$`v[i]`$\rfloor$. No final as listas são percorridas por ordem e os seus elementos são colocados em `v[]`.

```
  Algoritmo de ordenação bucket sort generalizado
  Vector a ordenar: v[1..n]; contém reais em [0,1)
  Vector auxiliar:  c[1..u] (cada elemento é uma lista ligada)
  O resultado continua em v[]
  int(x): parte inteira de x

 1   for i=1 to n:  c[i] = NIL
 2   for i=1 to n:
 3     insere v[i] no início da lista c[int(n*v[i])]
 4   for i=1 to n:
 5     ordena a lista c[i] pelo método de inserção (quadrático em n)
 6   k=1
 7   for i=1 to n:
 8     p=c[i]
 9     while p != NIL:
10       v[k] = valor apontado por p
11       k = k+1
12       p = p.next
```

Correcção do algoritmo

Se `v[i]` e `v[j]` são colocados (linhas `2-3`) na mesma lista `c[i]`, essa lista é ordenada (linha `5`) e esses 2 valores vão ocorrer por ordem no resultado, dado que as operações das linhas `7-12` não alteram as posições relativas dos elementos comuns a cada lista.

Suponhamos agora que `v[i]` e `v[j]` são colocados em listas diferentes, sejam `c[i']` e `c[j']`, respectivamente. Sem perda de generalidade supunhamos que `v[i]<v[j]`; como $\mathtt{i}' = \lfloor \mathtt{v[i]} \rfloor$ e $\mathtt{j}' = \lfloor \mathtt{v[j]} \rfloor$ e como, por hipótese $\mathtt{i}' \neq \mathtt{j}'$, é obrigatoriamente $\mathtt{i}' < \mathtt{j}'$. Logo, no final, `v[i]` e `v[j]` são colocados em posição relativa correcta, isto é, `v[i]` antes de `v[j]` se `v[i]¡v[j]` e `v[i]` depois de `v[j]` se `v[i]>v[j]`.

Eficiência do algoritmo

Vamos mostrar que o tempo médio do algoritmo é linear em n. As linhas `1-3` e `6-12` são executadas em tempo $O(n)$. O importante é analisar a contribuição das n ordenações (linhas `4-5`).

Seja n_i o número de valores que são colocados na lista `c[i]`; trata-se de variáveis aleatórias. Supondo que os elementos a ordenar estão uniformemente distribuídos no intervalo $[0, 1)$, a probabilidade de $n_i = x$ tem a distribuição binomial correspondente à probabilidade $1/n$ (probabilidade de um valor pertencer a uma lista fixa).

$$\begin{cases} \text{prob } (n_i = x) &= \binom{n}{x} p^x (1-p)^{n-x} \\ \mu &= np = n \times \frac{1}{n} = 1 \\ \sigma^2 &= np(1-p) = 1 - \frac{1}{n} \end{cases}$$

O valor médio do tempo total satisfaz

$$\begin{aligned} t(n) &\leq E(\textstyle\sum_i k n_i^2) && \text{(método da inserção é } O(n^2)) \\ &= kE(\textstyle\sum_i n_i^2) \\ &= k\textstyle\sum_i E(n_i^2) && \text{(linearidade de } E) \\ &= k\textstyle\sum_i [\sigma^2(n_i^2) + E(n_i)^2] && \text{(propriedade da variância)} \\ &= k\textstyle\sum_i \left(1 - \frac{1}{n} + 1\right) \\ &= kn\left(2 - \frac{1}{n}\right) \\ &\leq 2kn \end{aligned}$$

Logo $t(n)$ (tempo médio) é de ordem $O(n)$.

5.2 Métodos de ordenação baseados na representação dos valores

Nesta secção vamos supor que cada valor do vector está representado como uma sequência: por exemplo, uma sequência de dígitos – se os valores estiverem representados na base 10 – ou uma sequência de caracteres – se esses valores forem "strings".

Notemos que o comprimento de cada sequência é arbitrário. Notemos também que a definição da ordem entre os elementos do vector depende da sua natureza; por exemplo, segundo as convenções usuais é `15<132` mas `"15">"132"`.

5.2.1 "Radix sort": começando pelo símbolo mais significativo

Podemos ordenar os valores da seguinte forma

1. Separamos (ordenamos) os valores em grupos, segundo o símbolo (dígito ou caracter, por exemplo) mais significativo.
2. Cada grupo formado no passo 1. é ordenado pelo segundo símbolo mais significativo.

– ...

n. Cada grupo formado em (n-1). é ordenado pelo símbolo menos significativo.

O seguinte diagrama ilustra a ordenação de `[5211, 2233, 2122, 2231]`. Os valores estão escritos na vertical e os grupos formados estão marcados com "–".

```
         (1)          (2)          (3)          (4)
              -----  -     - ---  -     - ---  -     - - - -
5 2 2 2 -->  2 2 2 5      2 2 2 5      2 2 2 5      2 2 2 5
2 2 1 2      2 1 2 2 --> 1 2 2 2      1 2 2 2      1 2 2 2
1 3 2 3      3 2 3 1      2 3 3 2 --> 2 3 3 2      2 3 3 2
1 3 2 1      3 2 1 1      2 3 1 1      2 3 1 1 --> 2 1 3 1
```

5.2.2 "Radix sort": começando pelo símbolo menos significativo

A seguinte forma de ordenar está também correcta, mas é menos intuitiva; a primeira ordenação efectuada corresponde ao símbolo menos significativo e a última ao símbolo mais significativo.

1. Ordenamos os valores segundo o símbolo menos significativo, de forma estável.
2. Ordenamos os valores segundo o segundo símbolo menos significativo, de forma estável.

– ...

n. Ordenamos os valores segundo o segundo símbolo mais significativo, de forma estável.

O seguinte diagrama ilustra a ordenação de `[5211, 2213, 2122, 2223]`. Os valores estão escritos na vertical.

```
        (1)          (2)          (3)          (4)
5 2 2 2      5 2 2 2      5 2 2 2      2 5 2 2 --> 2 2 2 5
2 2 1 2      2 1 2 2      2 2 1 2 --> 1 2 2 2      1 2 2 2
1 1 2 2      1 2 1 2 --> 1 1 2 2      2 1 1 2      2 1 2 2
1 3 2 3  --> 1 2 3 3      1 3 2 3      2 1 3 3      2 3 3 1
```

Uma implementação na linguagem `python` deste algoritmo pode ver-se em apêndice, a página 173.

Este algoritmo é menos intuitivo, embora computacionalmente mais simples que o anterior; justifica-se a demonstração da sua correcção.

Correcção do algoritmo

Para vermos que este método está correcto raciocinemos por indução no número n de símbolos de cada valor (4 no exemplo anterior). O caso base, $n = 1$ é trivial. Suponhamos que $n \geq 2$ e que, pela hipótese indutiva, os valores já estão correctamente ordenados se ignorarmos o símbolo mais significativo (de ordem n). Sejam $x = ax'$ e $y = by'$ dois valores a ordenar, onde a e b representam símbolos e x, y, x' e y' representam sequências de símbolos.

1. Se $a < b$ é necessariamente $x < y$ e portanto a ordenação segundo o símbolo mais significativo vai colocar x antes de y no vector; conclusão análoga se obtém no caso $a > b$.

2. Se $a = b$, pela hipótese indutiva

   (a) se $x' < y'$, ax' ocorre antes de by' no vector e, como a ordenação segundo o símbolo de ordem n é estável e $a = b$, vai continuar no fim ax' antes de by'.

   (b) se $x' > y'$, conclui-se de modo semelhante que no fim ax' vai ficar depois de by' no vector.

Vemos que é essencial que, em cada nível de significância, as ordenações pelos símbolos sejam estáveis.

Eficiência do algoritmo

Sejam

n: o número de elementos do vector a ordenar

u: o tamanho do alfabeto, por exemplo 10 para os inteiros analisados dígito a dígito e 256 para as palavras analisadas caracter a caracter.

m: o maior comprimento de uma palavra presente no vector.

O algoritmo é de ordem $\Theta(mn + u)$; se m e u forem considerados constantes, o algoritmo é linear em n.

Exercício 53 *Analisando o programa apresentado em apêndice na página 173 (e supondo que as operações elementares das filas são $O(1)$) mostre que a ordem de grandeza do algoritmo é $\Theta(mn + u)$.*

Referência aos "Tries"

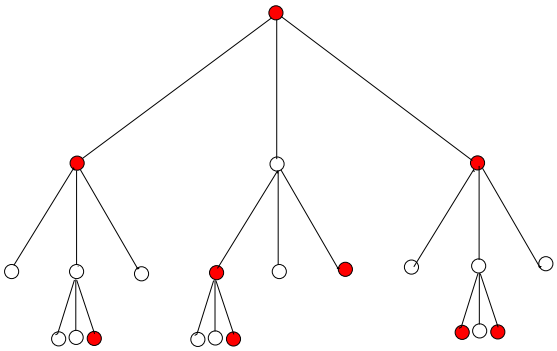
Podemos dizer que os algoritmos de ordenação descritos em (5.1.1), (5.1.3), (5.1.5), (5.2.1) e (5.2.2) são baseados na indexação pelos valores a ordenar (ou parte desses valores) e não por comparação entre esses valores. Existe uma estrutura de informação, a `trie` que é também baseada nos valores que se pretendem procurar, introduzir ou eliminar.

Num "trie" os valores não estão explicitamente registados, mas resultam do caminho – sequência de índices – desde a raíz até um determinado nó. Um "trie" definido num alfabeto finito Σ é uma árvore em que cada nó ou é uma folha (nó terminal) ou tem exactamente $|\Sigma|$ filhos indexados por Σ.

Por exemplo, a seguinte "trie" represente o conjunto

$$\{\varepsilon, \texttt{a}, \texttt{aco}, \texttt{ca}, \texttt{cao}, \texttt{co}, \texttt{oca}, \texttt{oco}\}$$

Os caracteres `a`, `c` e `o` são representados respectivamente por ramos (descendentes) para a esquerda, para baixo e para a direita. Por exemplo. a palavra `aco` corresponde ao seguinte caminho que parte da raíz: esquerda (`a`), baixo (`c`) e direita (`o`); o nó correspondente está marcado a vermelho.



Não vamos apresentar detalhes sobre o "trie", ver bibliografia.

5.3 Mediana; selecção

A mediana de um vector `v[1..n]` cujos elementos são todos distintos é o elemento do vector que tem $\lfloor n/2 \rfloor$ elementos menores que ele; por outras palavras, se o vector fosse ordenado por ordem crescente, a mediana seria o elemento de índice $1 + \lfloor n/2 \rfloor$. Exemplos: a mediana de `[6,8,1,5,4]` é `5`, e a mediana de `[2,6,8,1,5,4]` é também `5`.

Mais geralmente vamos considerar o problema da selecção do elemento de um vector que tem uma ordem dada i, o "i-gésimo" elemento.

Definição 6 Selecção e mediana
O elemento de ordem i de um vector `v` *com n elementos distintos é o elemento do vector que tem exactamente $i-1$ elementos menores que ele. O valor de i pode ir de 1 a n. A mediana de um vector* `v` *com n elementos distintos é o elemento de ordem $1 + \lfloor n/2 \rfloor$ desse vector.*

Um método de calcular a mediana é ordenar o vector e retornar $v[1 + \lfloor n/2 \rfloor]$. Mas este método não é muito eficiente, pois estamos a gastar um tempo de ordem $\Omega(n \log n)$ (a ordem de grandeza dos melhores algoritmos de ordenação) para se obter apenas um elemento do vector.

Exercício 54 *Mostre que o elemento de ordem 0 de um vector (mínimo) pode ser determinado em tempo $O(n)$. Mostre que o elemento de ordem 1 de um vector (o segundo menor elemento) pode ser determinado em tempo $O(n)$.*

Exercício 55 *O seguinte resultado está correcto? Porquê? Teorema (correcto?) Qualquer que seja m com $0 \leq m < n$ (m pode ser uma função de n como por exemplo $m = n/2$), o elemento de ordem m pode ser determinado em tempo de ordem $O(n)$.*
Dem. Por indução em m. O caso base $m = 0$ é trivial (mínimo do vector). Pela hipótese indutiva, suponhamos que se obtém o elemento de ordem $m-1$, seja x, em tempo de ordem $O(n)$, seja $t_1 \leq k_1 n$ para $n \geq n_1$. Para obter o elemento de ordem m, basta procurar o mais pequeno elemento do vector que é maior que x, o que obviamente pode ser efectuado em tempo $O(n)$, seja $t_2 \leq k_2 n$ para $n \geq n_2$. Considerando o tempo total $t = t_1 + t_2$ e tomando $k = k_1 + k_2$ e $n_0 = \max\{n_1, n_2\}$, chegamos à conclusão pretendida.

Nesta secção vamos mostrar os seguintes resultados

- O elemento de ordem i de um vector `v` com n elementos pode ser determinado em tempo médio $O(n)$.
- O elemento de ordem i de um vector `v` com n elementos pode ser determinado em tempo $O(n)$, mesmo no pior caso.

O segundo resultado é particularmente surpreendente. Durante muito tempo pensou-se que tal não era possível. E um algoritmo para o conseguir resultou de algum modo de uma tentativa para mostrar a sua impossibilidade!

Na prática, se se prevê que se vai executar o algoritmo de selecção mais que, digamos, $\log n$ vezes sem alterar o vector, pode ser preferível ordenar previamente o vector (usando um método de ordenação eficiente); num vector ordenado a selecção é trivial.

Tanto na Secção 5.3.1 como na Secção 5.3.2 vamos usar um procedimento de "`split(x)`", análogo ao utilizado no quick sort, que divide o vector em 2 partes

- Parte esquerda: elementos com valor inferior a x.
- Parte direita: elementos com valor superior a x.

Este procedimento determina também o número k de elementos da parte esquerda.

5.3.1 Mediana em tempo médio $O(n)$

Usa-se o procedimento "`split`" por forma a dividir o vector `v[1..n]` em 2 partes, aplicando-se recursivamente a função à parte esquerda do vector ou à parte direita do vector, até se determinar a mediana. Mais precisamente, a função para a obtenção do elemento de ordem i é a seguinte

```
Função para calcular o elemento de ordem i no vector v
SEL(i,v[a..b])
// na  chamada inicial: a=1, b=n e admite-se que a <= i <= b
   1) Escolhe-se x=v[a]  // outras escolhas para pivot são possíveis!
   2) split(x); Seja k o número de elementos na parte esquerda
   3) se i=k+1: return v[i]
      se i<k+1: SEL(i,      v[a..a+k-1])
      se i>k+1: SEL(i-(k+1),v[a+k+1..b])
```

A figura seguinte ilustra os índices do vector relacionados com a partição. Na linha de cima indicamos o tamanho dos sub-vectores e na de baixo os correspondentes índices.

k	1	$b-a-k$
$a, a+1 \ldots\ldots\ldots\ldots\ldots, a+k-1$	$a+k$	$a+k+1, a+k+2, \ldots\ldots\ldots\ldots\ldots, b$

Nota.

Análise do algoritmo, pior caso. É evidente que o tempo de execução no pior caso é de ordem $O(n^2)$; na verdade, pode acontecer que os sucessivos tamanhos dos lados esquerdos das divisões resultantes dos "splits" serem $n-1$, $n-2$,..., 1. Nesse caso, os número de comparações dos sucessivos "splits" são

$$(n-1) + (n-2) + \ldots + 1 = (n-1)n/2 \in \Omega(n^2)$$

Análise do algoritmo, caso médio. Quando é efectuado o "split" (linha 2) as duas partes podem ter, com igual probabilidade, os tamanhos

$$(n-1, 0),\ (n-2, 1),\ \ldots,\ (1, n-2),\ (0, n-1)$$

É claro que em cada um destes casos o lado (esquerdo ou direito) da chamada recursiva depende também da ordem i, parâmetro de chamada. Mas como pretendemos apenas calcular um majorante do caso médio, tomamos o maior dos lados em cada uma das n divisões. Vamos mostrar por indução em n que $E(t(n)) \leq 4n$. Por simplicidade vamos supor que n é par, de modo que o conjunto $\{n/2, n/2+1, \ldots, n-1\}$ tem exactamente $n/2$ elementos; o caso geral não apresenta dificuldade de maior.

$$\begin{aligned}
E(t(n)) &\leq (n-1) + \frac{2}{n}\sum_{i=n/2}^{n-1} E(t(i)) && (5.1)\\
&= (n-1) + E[E(t(n/2)) + E(t(n/2+1)) + \ldots + E(t(n-1))] && (5.2)\\
&\leq (n-1) + E(4(n/2) + 4(n/2+1) + \ldots + 4(n-1)) && (5.3)\\
&\leq (n-1) + 4 \times \frac{3n}{4} && (5.4)\\
&\leq 4n && (5.5)
\end{aligned}$$

Justificação:

– (5.1): Substituição para $k < n/2$ dos tamanhos k por $n - 1 - k$.

– De (5.1) para (5.2): A média de $n/2$ valores é a sua soma a dividir por $n/2$.

– De (5.2) para (5.3): Hipótese indutiva.

– De (5.3) para (5.4) e de (5.4) para (5.5): Propriedades simples.

Assim temos o seguinte resultado

Teorema 17 *Para qualquer $i \in \{1, 2, \ldots, n\}$ o algoritmo da página 90 tem tempo médio $O(n)$, efectuando um número de comparações não superior a $4n$.*

5.3.2 Mediana em tempo $O(n)$ (pior caso)

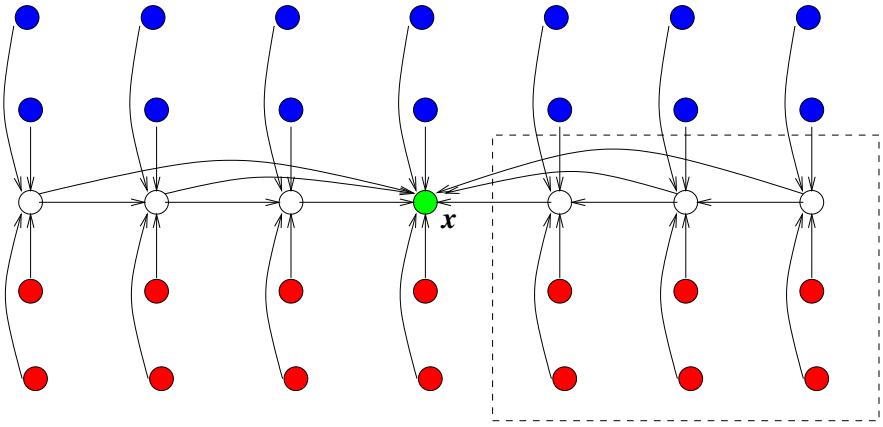
Para se conseguir um tempo de ordem $O(n)$ no pior caso é essencial conseguir obter um elemento x do vector que tem, qualquer que seja o vector com n elementos, pelo menos fn elementos menores que ele e pelo menos fn elementos maiores que ele, onde f é um número real positivo conveniente. Apresentamos em primeiro lugar um esquema do algoritmo e depois faremos a sua análise.

No algoritmo descrito em seguida[2] supomos por simplicidade que todas as divisões dão resto 0. O caso geral não é muito mais complicado, ver por exemplo [2].

```
Algoritmo para seleccionar o elemento de ordem i no vector v
SEL(i,v[a..b])   // na  chamada inicial: a=1, b=n
   1) Dividem-se os n elementos do vector em n/5 grupos de 5
      elementos.
   2) Determina-se a mediana de cada um desses grupos de 5
      elementos.
   3) Chama-se recursivamente SEL para determinar a mediana x das
      n/5 medianas
   4) Faz-se um "split" de v, usando x como pivot
      Seja k o número de elementos no lado esquerdo (<x)
      e n-k no lado direito (>x)
   5) se i=k+1: return v[i]
      se i<k+1: SEL(i,      v[a..a+k-1])
      se i>k+1: SEL(i-(k+1),v[a+k+1..b])
```

A figura seguinte representa o estado do conhecimento sobre a relação de ordem entre os elementos do vector após o "split" (instrução 4); Uma seta de um valor a para um valor b significa que é forcosamente $a < b$.

[2] O número 5 que aparece no algoritmo não é "mágigo"; obteríamos o mesmo resultado com divisões em grupos de tamanho maior que 5.



Análise do algoritmo Observemos em primeiro lugar que, conforme resulta da figura anterior, o número de elementos do vector que excedem x, a mediana das medianas, é pelo menos (no figura: elementos dentro do rectângulo tracejado): número de grupos de 5 elementos à direita de x (há 3 desses grupos na figura) vezes 3 (estamos a desprezar 2 elementos do grupo de 5 a que x pertence), ou seja[3]

$$\frac{n/5-1}{2} \times 3 \quad \in \Omega(n)$$

(cerca de $3n/10$ elementos). No exemplo da figura teríamos o valor $(6/2) * 3 = 9$, o número de elementos vermelhos ou brancos à direita (mas não por baixo) de x. Assim, na chamada recursiva da instrução 5, o tamanho do sub-vector é, no máximo,

$$n - 3 \times \frac{n/5-1}{2} = n - \frac{3n}{10} + \frac{3}{2} = \frac{7n}{10} + \frac{3}{2}$$

A constante 3/2 pode ser ignorada, incorporando-a na constante c do termo geral da recorrência (5.6). Vamos agora escrever uma recorrência que permite obter um majorante para o tempo do algoritmo. Seja $t(n,i)$ o tempo, no pior caso, para determinar o elemento de ordem i de um vector com n elementos e seja $t(n) = \max_i\{t(n,i)\}$. Temos as seguintes contribuições para o majorante de $t(n)$:

- Passos 1 e 2: $k_1 n$ (note-se que a mediana de um conjunto de 5 elementos se pode obter em tempo constante.
- Passo 3: $t(n/5)$
- Passo 4: $k_2 n$

[3]Uma análise geral, onde não se supõe que as divisões são exactas, permite obter o minorante $\frac{3n}{10} - 6$ para do número de elementos à direita (ou à esquerda de x). Isto quer dizer que, na chamada recursiva da instrução 5, o tamanho do sub-vector é, no máximo, $n - [(3n/10) - 6] = (7n/10) + 6$.

– Passo 5: $t(7n/10)$

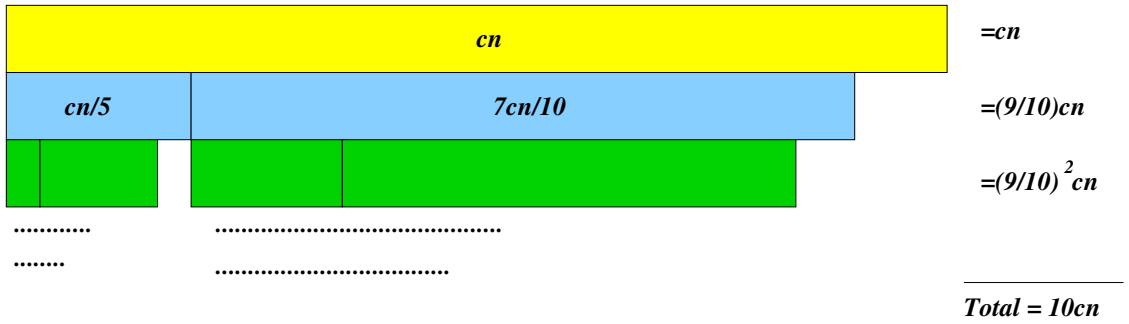
O termo geral da recorrência é

$$t(n) \leq cn + t(n/5) + t(7n/10) \tag{5.6}$$

onde $c = k_1 + k_2$. Para resolver esta recorrência, ou melhor, para obter um majorante da sua solução, vamos "usá-la" repetidamente

$$\begin{aligned}
t(n) &\leq cn + t(n/5) + t(7n/10) \\
&\leq cn + c(n/5) + c(7n/10) + [t(n/25) + t(7n/50)] + [t(7n/50) + t(49n/100)] \\
&= cn(1 + (9/10)) + [t(n/25) + t(7n/50)] + [t(7n/50) + t(49n/100)] \\
\ldots &\quad \ldots \\
&\leq cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \ldots) \\
&= 10cn
\end{aligned}$$

Assim, $t(n)$ é de ordem $O(n)$. O raciocínio usado neste desenvolvimento está ilustrado na figura seguinte



Resumindo,

Teorema 18 *Para qualquer $i \in \{1, 2, \ldots, n\}$ o algoritmo 5.3.1 tem tempo no pior caso de ordem $O(n)$.*

Usando a ideia da demonstração anterior não é difícil demonstrar o seguinte resultado.

Teorema 19 (Recorrências com solução $O(n)$) *Seja a equação geral de uma recorrência*

$$f(n) = f(k_1 n) + f(k_2 n) + \ldots + f(k_p n) + cn$$

onde $c \geq 0$ *e* k_1, k_2,..., k_p *são constantes positivas com* $k_1 + k_2 + \ldots + k_k < 1$. *Então* $f(n)$ *é de ordem* $O(n)$.

Este resultado deve ser junto à nossa "lista de resultados" sobre a resolução de recorrências; essa lista inclui o Teorema 10.

Capítulo 6

Circuitos e redes de ordenação

As redes de ordenação são um tipo particular de *circuitos*. Começamos por apresentar alguns conceitos genéricos sobre circuitos.

6.1 Circuitos

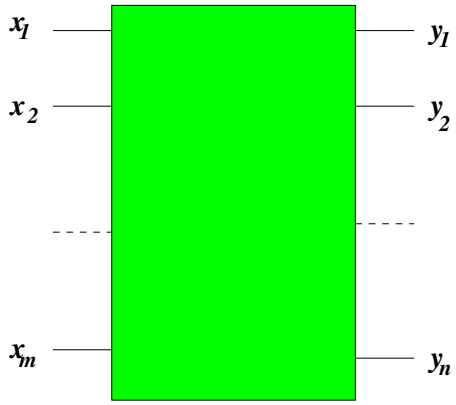
Os circuitos são uma alternativa não uniforme aos algoritmos. Enquanto um algoritmo pode ter dados de qualquer comprimento, um circuito tem uma aridade (número de argumentos) fixa. Assim, por exemplo, uma alternativa a um algoritmo de ordenação é uma família infinita C_1, C_2,... de circuitos em que o circuito C_n ordena vectores com n elementos. Pode não existir um método único de construir circuitos de uma família – o modelo dos circuitos não é, em princípio, uniforme sendo por isso computável[1].

Consideremos um domínio D em que vamos definir um circuito; esse domínio pode ser, por exemplo $\mathbb{N}$ ou o conjunto dos valores lógicos, $\mathbb{B} = \{F, V\}$. Um *componente* (ou circuito elementar) c com m entradas e n saídas é uma sequência de funções

$$c_1(x_1, \ldots, x_m),\ c_2(x_1, \ldots, x_m), \ldots,\ c_n(x_1, \ldots, x_m)$$

Este componente pode ser representado por (figura do lado esquerdo)

[1] Tal facto coloca questões a nível da computabilidade e de complexidade das computações. Por exemplo, a função implementada por cada circuito tem um número fixo de argumentos, sendo por essa razão computável; mas a função implementada pela família de circuitos pode não o ser.



Definição 7 *Um* circuito *com m entradas e n saídas definido num domínio D e com componentes de uma família $\mathcal{F}$ é um grafo dirigido acíclico com 3 tipos de nós*

- *m nós de entrada*
- *n nós de saída*
- *Componentes*

Os arcos podem ligar, pela ordem indidada, os seguintes tipos de nós

- *nós de entrada a nós de saída*
- *nós de entrada a entradas de componentes*
- *saídas de componentes a entradas de (outros) componentes*
- *saídas de componentes a nós de saída*

Tanto os nós de saída do circuito como os nós de entrada nos componentes têm grau de entrada 1. Por outras palavras, não se pode ligar ao mesmo nó mais que um valor fixado pelo utilizador ou pela saída de um componente (grau de entrada ≤ 1) e tem que se definir o valor de todos esses nós (grau de entrada ≥ 0).

A cada nó de um circuito podemos atribuir uma *profundidade* que corresponde ao número de passos necessários para calcular (em paralelo) o valor nesse nó. Mais precisamente,

Definição 8 *A* profundidade *de um nó z de um circuito com m entradas e n saídas é*

$$\text{depth}(z) = \begin{cases} 0 & \text{se } z \text{ é um nó de entrada} \\ \max\{1 + \text{depth}(w_i) : i = 1, 2, \ldots, k\} & \text{se } z \text{ é a saída de um componente} \end{cases}$$

No último caso o componente é $c(w_1, w_2, \ldots, w_k)$. A profundidade de um circuito é a máxima profundidade dos nós de saída.

A profundidade mede o tempo paralelo de execução; é fácil ver-se que, se cada componente demorar uma unidade de tempo a calcular o valor das saídas, o tempo total de execução é exactamente a profundidade do circuito. Por outro lado, o número de componentes mede de algum modo a complexidade espacial da computação. *Note-se que, para cada circuito, estes valores são fixos.*

Em resumo, num circuito temos a correspondência

Tempo de execução paralelo	=	Profundidade
Tempo de execução sequencial	=	Número de componentes

Uma família de componentes diz-se completa para uma classe de transformações se todas as transformações da classe podem ser implementadas usando unicamente componentes dessa família. Se $\mathcal{F}_1$ é uma família completa, para mostrar que $\mathcal{F}_2$ também é uma família completa, basta mostrar que cada componente de $\mathcal{F}_1$ pode ser implementado com um circuito que usa só componentes de $\mathcal{F}_2$.

Um circuito corresponde a um sistema de equações em que cada equação corresponde a um componente (equação da forma $z = c(w_1, w_2, \ldots, w_k)$) ou a uma ligação da entrada à saída (equação da forma $y_j = x_i$). Esse sistema pode ser resolvido de forma a expreesar as saídas y_j como função apenas das entradas x_i

$$y_i = f_i(x_1, x_2, \ldots, x_m) \ \ (1 \leq i \leq n)$$

a profundidade do circuito é o máximo grau de parêntização destas expressões.

Exercício 56 *Desenhe um circuito no domínio* $\mathbb{B}$ *com família* $\{\neg, \wedge, \vee\}$ *que implemente a função* XOR *(o circuito tem 2 entradas e uma só saída). Diga qual a profundidade e o número de componentes do circuito. Usando o circuito que desenhou, escreva uma expressão funcional correspondente ao* XOR*.*

Exercício 57 *Supondo que a família* $\{\neg, \wedge, \vee\}$ *é completa para a classe de todas as transformações proposicionais, mostre que a família constituída apenas pelo circuito elementar* NAND *também é completa nessa classe. Nota:* NAND(x, y) *tem o valor* Falso *sse* x *e* y *têm ambos o valor* Verdade*.*

Exercício 58 *Mostre como definir para todo o* $n \geq 1$ *um circuito que implementa a função paridade*

$$\text{par}(x_1, x_2, \ldots, x_n) = \begin{cases} 0 & \text{se o número de 1's em } \{x_1, x_2, \ldots, x_n\} \text{ é par} \\ 1 & \text{se o número de 1's em } \{x_1, x_2, \ldots, x_n\} \text{ é ímpar} \end{cases}$$

Use apenas o componente XOR*.*

Nota. O circuito é um modelo de computação muito utilizado em complexidade e em particular para o estabelecimeno de minorantes de complexidade.

Em (6.2) estudaremos um tipo muito particular de circuitos, as redes de ordenação.

6.1.1 Classes de complexidade associadas ao modelo dos circuitos

Não incluído

6.2 Redes de comparação e redes de ordenação

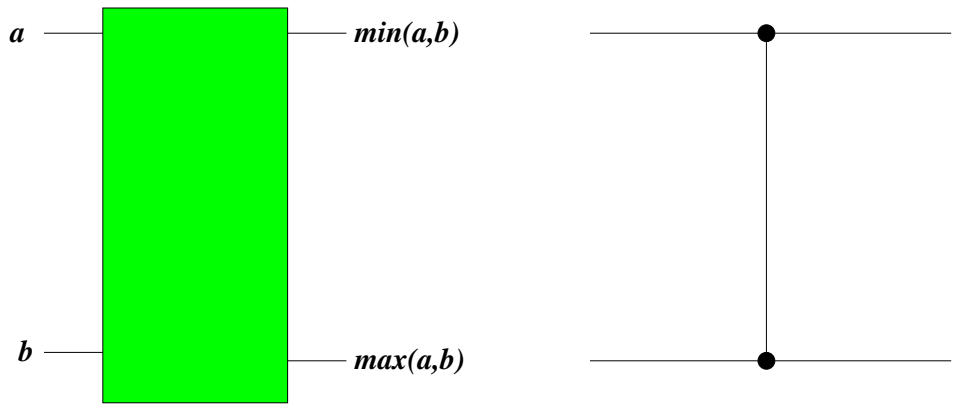
Plano desta secção
O objectivo principal desta secção é implementar uma[2] rede de ordenação eficiente. Essa rede será baseada no "merge sort". Para tal fim seguiremos os seguintes passos

- (6.2.1), página 98: introduzimos os conceitos fundamentais e caracterizamos uma família relativamente ineficiente de redes de ordenação que tanto pode ser inspirada no método clássico da ordenação por inserção como no da bolha ("bubble sort") (exercícios 61 e 62).
- (6.2.2), página 102: Demonstramos que para confirmar que uma rede de comparadores é também de ordenação basta analisar entradas só com 0's e 1's (Princípio 0/1).
- (6.2.3), página 103: Estudamos um tipo particular de ordenadores que funciona apenas para entradas "bitónicas"; para a construção destes ordenadores definiremos uma rede chamada "HC" ("half-cleaner"), que vai ser útil mais tarde.
- (6.2.4), página 105: Usaremos as redes HC para implementar a operação de "merge"; as redes de ordenação baseadas no "merge sort" resultarão trivialmente das redes de "merge".
- (6.2.5), página 107: Analisaremos a eficiência das diversas redes de ordenação estudadas e apresentaremos anguns minorantes de complexidade.

Para uma informação mais completa o leitor poderá consultar [2] e, sobretudo, a Secção 5.3.4 de [6].

6.2.1 Introdução e conceitos fundamentais

Uma *rede de comparação* (ou rede de comparadores) é um circuito em que o único componente é o seguinte (representado de forma esquemática à direita); o número de saídas é forcosamente igual ao número de entradas.

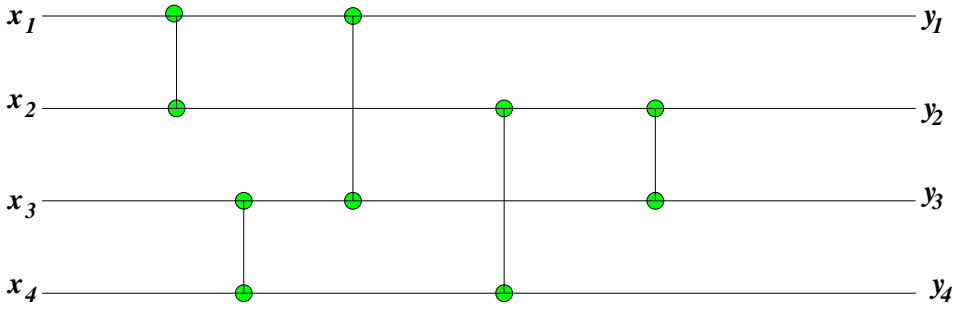


As saídas são: o menor e o maior dos valores de entrada; não sendo indicado expressamente o contrário, o menor dos valores fica em cima e o maior em baixo (como na figura anterior). Estes componentes são representados por "|" (como na figura anterior à direita) ou "↓". Quando pretendermos que o maior dos valores fique em cima (e o menor em baixo) usaremos o símbolo "↑".

Uma *rede de ordenação* é uma rede de comparação que ordena os valores por ordem crescente; por outras palavras, quaisquer que sejam os valores de entrada, a saída contém esses mesmos valores de forma ordenada, crescendo de cima para baixo.

Observação. As redes de ordenação também se chamam "ordenadores com esquecimento ("oblivious sorters"). O nome provém do facto de os comparadores actuarem em linhas fixas da rede, independentemente das comparações anteriores. Uma forma mais geral de ordenadores utiliza testes, correspondendo a árvores de ordenação; por exemplo, na raíz da árvore compara-se `v[3]` com `v[5]`; se for `v[3]<v[5]`, vai-se comparar `v[2]` com `v[8]`; senão... Como se compreende, este tipo de ordenadores é mais difícil de paralelizar. (fim da observação)

Considere a seguinte rede de comparação:

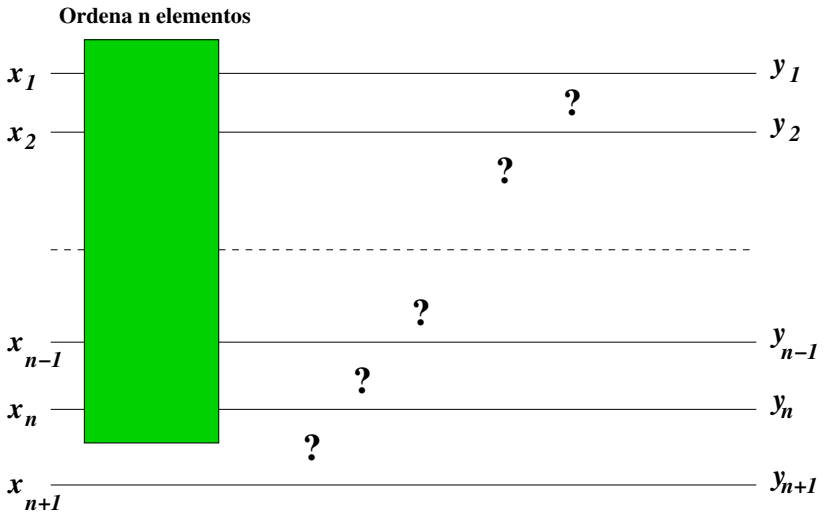


Exercício 59 *Considere as entradas $x_1 = 8$, $x_2 = 5$, $x_3 = 1$, $x_4 = 6$.*

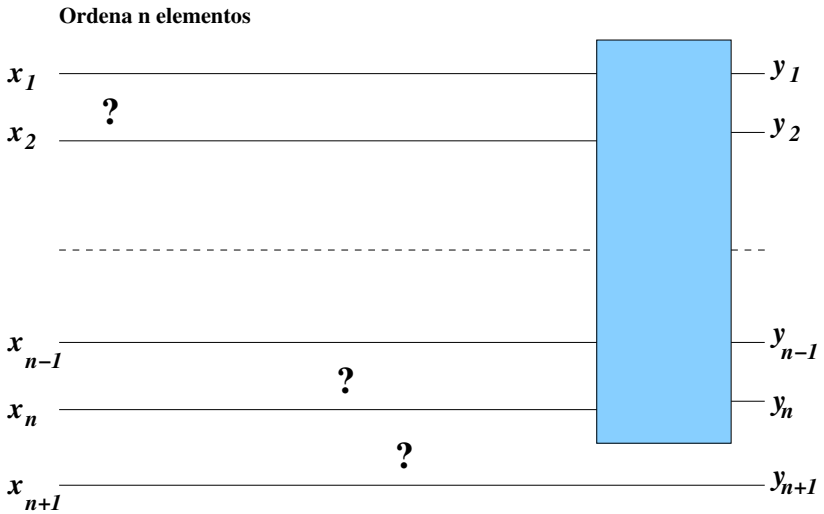
1. *Quais os valores de y_1, y_2, y_3 e y_4?*
2. *Qual é a profundidade do circuito?*
3. *Quantas comparações são efectuadas?*
4. *Indique os valores que ficam definidos ao fim de t unidades de tempo com $t = 0, 1, 2, \ldots$.*

Exercício 60 *Mostre que a rede de comparação mencionada no exercício anterior é uma rede de ordenação.*

Exercício 61 *Mostre que para todo o $n \in \mathbb{N}$ existe uma rede de ordenação. A sua demonstração deverá usar indução em n. Considere a figura em baixo; use o princípio da indução para um "design" do circuito baseado no método de ordenação por inserção: pela hipótese indutiva, o bloco ordena de forma correcta n elementos. Como se poderá inserir na posição correcta o elemento $n+1$?*



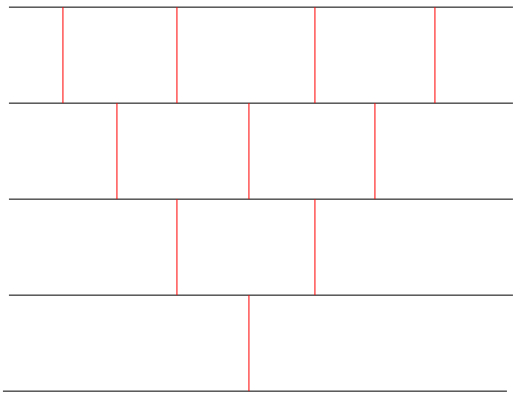
Exercício 62 *Implemente uma rede de ordenação baseada no método de ordenação da bolha ("bubble sort"). Use um método semelhante ao do exercício anterior, começando contudo por colocar o maior elemento na linha $n+1$.*



Exercício 63 *Desenhe o circuito correspondente ao método do exercício 61 para o caso $m = 4$ (ordenação de 4 elementos). Repita para o exercício 62. Mostre que os circuitos são o mesmo!*

As redes de ordenação definidas nos exercícios 61 (baseada no método da inserção) e 62 (baseada no método ba bolha) não são muito eficientes, nem em tempo nem em espaço. No exercício seguinte mostra-se que a profundidade (tempo) é de ordem n enquanto a complexidade espacial é $O(n^2)$.

Exercício 64 *Mostre que o tempo de ordenação do circuito baseado no método de inserção (ou no da bolha, dado que são iguais) é $2n - 3$. Mostre que o número de comparadores é $n(n-1)/2$.*



Posteriormente caracterizaremos circuitos que ordenam de forma mais eficiente. A seguinte tabela faz um sumário[3] da eficiência dos diversos métodos. Recorda-se que, no modelo algorítmico, nenhum algoritmo de ordenação baseado em comparações efectua menos que (e portanto tem uma eficiência temporal melhor que) $\log(n!) \in \Theta(n \log n)$.

Método	Profundidade	Número de componentes
Inserção	$\Theta(n)$	$\Theta(n^2)$
Bolha	$\Theta(n)$	$\Theta(n^2)$
Baseado no "merge"	$\Theta(\log^2(n))$	$\Theta(n \log^2(n))$
Baseado no "Shell sort"	$\Theta(\log^2(n))$	$\Theta(n \log^2(n))$
AKS	$\Theta(\log(n))$	$\Theta(n \log(n))$

[3]Com $\log^2(x)$ pretendemos designar $(\log x)^2$ (e não $\log(\log x)$).

6.2.2 Princípio 0/1

Vamos demonstrar um resultado muito útil na análise e projecto de redes de ordenação: se uma rede de ordenação ordena correctamente qualquer sequência de 0's e 1's, então ordena correctamente qualquer sequência de números reais. Usando este resultado, para verificar que uma rede de comparadores é uma rede de ordenação, basta analisar 2^n sequências de entradas em vez de $n!$ (ou ainda mais (quanto?) se admitirmos a existência de elementos repetidos). A diferença entre os dois valores é enorme.

Exercício 65 *Verifique que $n!/2^n$ cresce mais rapidamente que qualquer potência a^n (com $a \geq 0$), provando que*

$$\forall a \geq 0 : \lim_{n\to\infty} \frac{n!/2^n}{a^n} = +\infty$$

Definição 9 *Uma função $f(x)$ diz-se* monótona *se $x \leq x'$ implica $f(x) \leq f(x')$; em particular, as funções constantes são monótonas.*

Lema 1 *Se f é uma função monótona, uma rede de comparadores que transforme $x = x_1x_2\ldots x_n$ em $x = y_1y_2\ldots y_n$, transforma $f(x) = f(x_1)f(x_2)\ldots f(x_n)$ em $f(y) = f(y_1)f(y_2)\ldots f(y_n)$.*

Dem. Usamos indução na profundidade d do circuito. Se $d = 0$, não há comparadores e a prova é trivial. Consideremos o caso $d > 0$ e um comparador cujas saídas estão à profundidade d. Suponhamos que é aplicado x à entrada e sejam z_1 e z_2 as entradas do comparador e y_i e y_j as saídas

$$\begin{cases} y_i = \min(z_1, z_2) \\ y_j = \max(z_1, z_2) \end{cases}$$

Pela hipótese indutiva (à esquerda do circuito a profundidade é inferior a d), se aplicarmos $f(x)$ à entrada do circuito, as entradas do comparador são $f(z_1)$ e $f(z_2)$. Mas então, e usando a monotonia de f, temos as saídas

$$\begin{cases} \min(f(z_1), f(z_2)) = f(\min(z_1, z_2)) \\ \max(f(z_1), f(z_2)) = f(\max(z_1, z_2)) \end{cases}$$

□

Teorema 20 *(Princípio 0/1) Se uma rede de ordenação ordena correctamente qualquer sequência de 0's e 1's, então ordena correctamente qualquer sequência de números reais*[4].

[4]O nome deste resultado, "Princípio 0/1", é usado em Combinatória com um significado completamente diferente

Este princípio também se aplica ao "merge": o "merge" correcto de sequências arbitrárias de 0's e 1's implica , o "merge" correcto de números reais[5].

Dem. Por contradição. Suponhamos que a rede ordena todas as sequências de 0's e 1's, mas que existe uma sequência de reais na entrada $x = x_1 x_2 \ldots x_n$ tal que $x_i < x_j$ mas x_j ocorre antes (acima) de x_i na saída. Defina-se a função monótona f por: $f(z) = 0$ se $z \leq x_i$ e $f(z) = 1$ se $z > x_i$. Pelo Lema anterior, uma vez que na saída x_j ocorre antes de x_i, também $f(x_j) = 1$ ocorre antes de $f(x_i) = 0$ e, portanto, a rede não ordena correctamente todas as sequências de 0's e 1's, o que é uma contradição. □

Exercício 66 *Usando o Princípio 0/1, mostre que a rede de comparadores da página 99 é uma rede de ordenação.*

6.2.3 Ordenadores bitónicos

Usando o Princípio 0/1, vamos caracterizar redes de ordenação bastante eficientes em tempo e espaço[6], os ordenadores bitónicos. Estas redes são ordenam apenas um tipo muito particular de sequências, as sequências "bitónicas"; servem contudo como base para a construção de ordenadores genéricos.

Definição 10 *Uma sequência de 0's e 1's diz-se* bitónica *se é da forma* $0^i 1^j 0^k$ *ou* $1^i 0^j 1^k$ *para inteiros não negativos* i, j *e* k *O conceito pode facilmente generalizar-se às sequências de números reais.*

Exemplos de sequencias bitónicas: 0000110, 000, 11110, 1101, 12342. Exemplos de de sequencias não bitónicas: 0100001, 10110, 123425.

Para construir o ordenador bitónico vamos primeiro definir uma rede conhecida por HC ("half-cleaner") que transforma qualquer sequência bitónica x (de 0's e 1's) de comprimento $2n$ em 2 sequências y e z de comprimento n com as seguintes propriedades (não independentes, 3 é consequência de 2):

1. y e z são bitónicas
2. Ou y tem só 0's ou z tem só 1's (ou as 2 coisas)

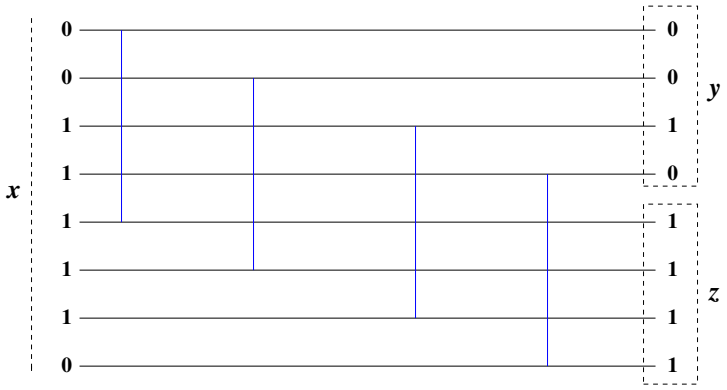
que é, em termos gerais, o seguinte: para valores de um determinado parâmetro inferiores a um certo valor a, a probabilidade assimptótica de uma determinada propriedade ocorrer é 0, sendo essa probabilidade 1 se o parâmetro for maior que a (ou o inverso).

[5] A demonstração apresentada aplica-se apenas à operação de ordenação.

[6] Recordamos que o espaço dos circuitos (número de componentes) reflete essencialmente o tempo algorítmico (sequencial).

3. Qualquer elemento de y é menor ou igual que qualquer elemento de z

A construção da rede HC é muito simples: existem comparadores entre as linhas 1 e $n+1$, 2 e $n+2$,..., n e $2n$, conforme se ilustra na figura seguinte para o caso $2n = 8$ (Figura de [2])



Exercício 67 *Mostre que com esta construção se verificam a propriedades 1, 2 e 3 que caracterizam as redes HC.* **Sugestão.** *Considere o caso $x = 0^i 1^j 0^k$ (o outro caso é semelhante) com $|x| = 2n$ e analise separadamente os diversos casos: a sequência de 1's de x*

1. *está nos primeiros n bits de x*
2. *está nos últimos n bits de x*
3. *intersecta a primeira e a segunda parte de x e $j \leq n$*
4. *intersecta a primeira e a segunda parte de x e $j > n$*

Construir um ordenador bitónico (para entradas bitónicas) usando redes HC é quase um exercício de projecto baseado no princípio da indução.

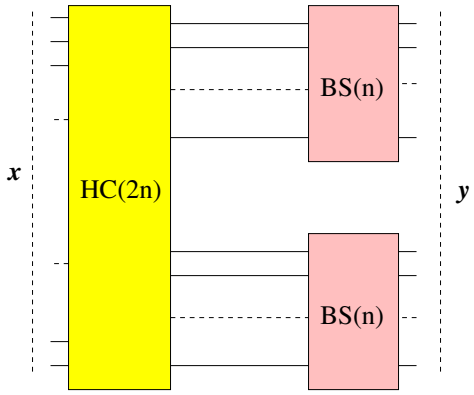
Vamos caracterizar o ordenador bitónico (BS, "bitonic sorter") de forma indutiva, mostrando ao mesmo tempo, por indução, que o circuito construído ordena qualquer sequência de 0's e 1's. O BS(n) vai ser caracterizado para valores de n que são potências de 2.

Dem. Caso Base, $n = 1$

O circuito não tem qualquer comparador, a transformação é a identidade, $y_1 = x_1$.

Tamanho $2n$, supondo que existe BS(n)

O circuito BS($2n$) é constituído por por um HC($2n$) à entrada, seguido por dois BS(n), um na parte superior, outro na parte inferior.

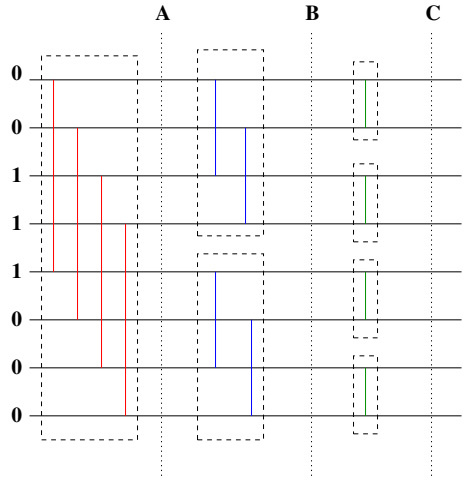


Dado que

- A entrada x é bitónica
- As 2 partes da saída do HC($2n$) são bitónicas e qualquer elemento da primeira parte é menor ou igual que qualquer elemento da segunda parte
- Os 2 BS(n) ordenam correctamente as entradas (pela hipótese indutiva)

concluimos que a saída y é a ordenação correcta da entrada x. □

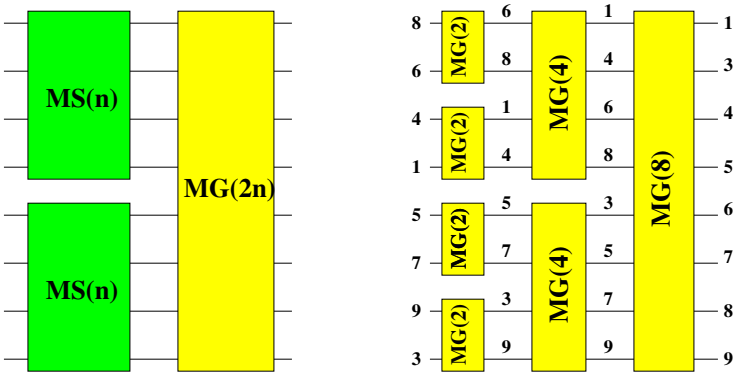
Exercício 68 *A figura seguinte descreve completamente o BS(8). Explique como foi obtida à custa da regra recursiva de construção de BS(n), ver a prova anterior. Indique os valores nas linhas verticais A, B e C.*



6.2.4 Rede de ordenação baseada no "merge sort"

Vamos finalmente descrever uma rede de ordenação muito eficiente; esta rede é baseada no "merge sort" clássico. Em primeiro lugar vamos descrever uma rede de "merge"; as redes de ordenação correspondentes constroem-se de forma recursiva muito simples, como passamos a explicar.

Regra recursiva: Seja MS(n) o circuito "merge sort" com n entradas e MG(n) o circuito de "merge" com n entradas. Um MS($2n$) constrói-se com duas redes MS(n) e uma rede MG($2n$), conforme se ilustra na figura seguinte (do lado esquerdo) para o caso $2n = 8$. À direita, a definição de MS foi completamente expandida, por forma a se obter uma rede constituída apenas por MG's.



É claro que necessitamos ainda de implementar a rede de "merge"!

Rede de "merge"

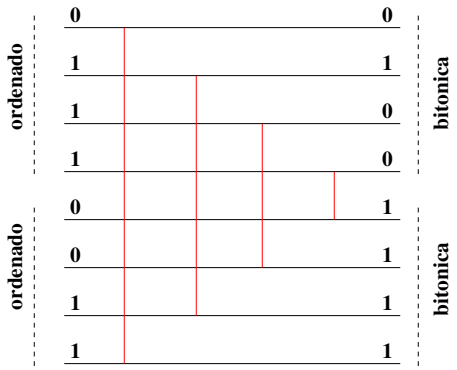
Vamos construir uma rede que recebe uma entrada de $2n$ valores (0 ou 1) constituída por 2 partes, os primeiros n e os últimos n valores, cada uma das quais ordenadas e produz à saída o resultado do "merge" dessas 2 partes. Vamos trabalhar apenas com 0's e 1's; o Princípio 0/1 garante-nos que a rede funcionará correctamente para quaisquer valores reais.

Reparemos que

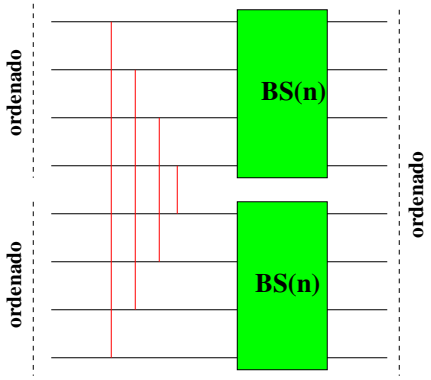
- Uma sequência ordenada é necessariamente da forma 0^i1^j, portanto bitónica
- O "merge" das sequências $w = 0^i1^j$ e $z = 0^k1^l$ é $0^{i+k}1^{j+l}$
- Se w e z estão ordenados, a aplicação da rede HC a wz^t (onde z^t representa z escrito "ao contrário") produz 2 sequências bitónicas em que todo o elemento da primeira parte é menor ou igual que todo o elemento da segunda parte. Por exemplo

$$\text{HC}(0111,\ 0011^t) \Rightarrow \text{HC}(0111,\ 1100) \Rightarrow \underline{0100},\ \underline{1111}$$

Verificamos que basta aplicar HC à entrada,invertendo a ordem dos elementos da segunda parte; essa inversão é implementada nas ligações dos comparadores, comparar com a figura da rede HC standard, página 104.



Temos 2 sequências bitónicas à saída; as propriedades 1 e 3 (página 103) dos circuitos HC são fundamentais: podemos ordenar as 2 partes da saída de forma independente, usando ordenadores bitónicos e temos o circuito "merge" genérico[7]!



6.2.5 Sumário, complexidade e minorantes

Estudamos em seguida a complexidade espacial e temporal das diversas redes de comparadores que estudamos – HC, ordenação bitónica, "merge" e "merge sort" – através de uma sequência de exercícios que o leitor deverá resolver. Alguns destes resultados estão sumariados na tabela da página 101. Seja

- MS(n): rede de ordenação baseada no "merge sort"; o número de entradas é n.
- MG(n): rede de ordenação que efectua um "merge" das primeiras $n/2$ com as segundas $n/2$ entradas; assume-se que n é par
- HC(n): "Half cleaner" com n entradas; assume-se que n é par
- BS(n): ordenador bitónico ("bitonic sorter") com n entradas

Nestes exercícios pode supor que o número de entradas n é uma potência de 2.

[7] Onde BS(n) representa o ordenados bitónico.

Exercício 69 Explique indutivamente como um MS(n) pode ser construído com base em MS(m) com $m < n$ e em redes MG.
Desenhe um MS(8) assumindo apenas a existência de MG's.

Exercício 70 Explique indutivamente como um MG(n) pode ser construído com base em BS(m) com $m < n$ e em redes HC.
Desenhe MG(4) e MG(8) assumindo apenas a existência de HC's.

Exercício 71 Explique indutivamente como um BS(n) pode ser construído com base em BS(m) com $m < n$ e em redes HC.
Desenhe BS(4) e BS(8) assumindo apenas a existência de HC's.

Exercício 72 Desenhe HC(2), HC(4), e HC(8) de forma explícita (sem usar outras redes).

Exercício 73 Combinando os resultados dos exercícios anteriores desenhe um MS(8) de forma explícita (sem usar outras redes).

Exercício 74 Foi estudada uma rede de ordenação baseada na ordenação clássica por inserção. Determine, como função do número de linhas n, a respectiva complexidade espacial (número de comparadores) e complexidade temporal (profundidade).

Exercício 75 Determine a ordem de grandeza exacta do número de comparadores da rede MS(n).
Sugestão. Considere sucessivamente o número de comparadores das redes BS, MG e MS.

Exercício 76 Determine a ordem de grandeza exacta da profundidade da rede MS(n).

Sugestão. Considere sucessivamente a profundidade das redes BS, MG e MS.

Exercício 77 Mostre que qualquer rede de ordenação tem um número de comparadores que é de ordem $\Omega(n \log n)$.

Sugestão. Considere a implementação sequencial da rede.

Exercício 78 Mostre que qualquer rede de ordenação tem uma profundidade de ordem $\Omega(n \log n)$.

Sugestão. Comece por verificar que qualquer saída y_k da rede, depende de todas as entradas, no sentido em que, qualquer que seja j com $1 \leq j \leq n$, e quaisquer que sejam as entradas x_i para $i \neq j$, existem valores x'_j e x''_j da entrada x_j tais que

$$y_k(x_1, \ldots, x_{j-1}, x'_j, x_{j+1}, \ldots, x_n) \neq y_k(x_1, \ldots, x_{j-1}, x''_j, x_{j+1}, \ldots, x_n)$$

Depois mostre que, se houver menos que $\log n$ comparadores, pelo menos uma das linhas x_i está "desligada" de y_j.

Na tabela seguinte usamos a notação

n: Número de linhas da rede de ordenação

d_m: Melhor minorante conhecido para a profundidade de uma rede de ordenação com n linhas.

d_M: Melhor majorante conhecido para a profundidade de uma rede de ordenação com n linhas.

c_m: Número mínimo de comparadores de uma rede de ordenação com n linhas.

TI: menor i tal que $2^i \geq n!$ (minorante do número de comparações baseado na Teoria da Informação) com n linhas.

Alguns destes resultados demoraram anos de investigação!

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
d_m	0	1	3	3	5	5	6	6	7	7	7	7	7	7	7	7
d_M	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9
c_m	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60
TI	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45

Capítulo 7

"Hash" universal e perfeito

Neste capítulo vamos estudar alguns aspectos importantes sobre os métodos de "hash" e, em especial o "hash" universal e "hash" perfeito. Os métodos de "hash" são de certo modo uma generalização dos algoritmos baseados na indexação (ver a Secção 5.1.2, página 80) permitindo tempo médio $O(1)$ nas operações de dicionário[1]. Mais especificamente, pretende-se construir uma função (de "hash") que transforme elementos de um universo arbitrário em inteiros positivos relativamente pequenos que indexem uma tabela – a tabela de "hash" – generalizando-se assim a operação de indexação. Com o método tradicional de "hash" consegue-se obter tempos de pesquisa, inserção e eliminação constantes (o que é excelente), mas apenas em termos do tempo médio – não no pior caso.

Pré-requesitos. Pressupomos que o leitor tem já um conhecimento genérico dos métodos de "hash" e da sua eficiência; neste capítulo vamos focar a nossa atenção em alguns tópicos mais avançados.

7.1 Considerações gerais sobre os métodos de "hash"

Nesta Secção vamos rever alguns dos conceitos ligados aos métodos de "hash".

7.1.1 Universos grandes, funções de "hash"

Os dicionários são uma das estruturas básicas utilizadas em programação e a sua implementação eficiente é de importância fundamental. Podemos considerar 2 tipos de dicionários:

– Dicionários estáticos: dado um conjunto N de elementos, pretendemos representá-los numa

[1]Neste Capítulo supomos que a função de "hash" é calculada em tempo constante.

estrutura de informação que permita uma procura eficiente. São aplicações possíveis o dicionário das palavras reservadas de uma determinada linguagem de programação e o dicionário que contém as palavras correctamente escritas de uma lingua (por exemplo, do Português).

Uma implementação razoável é um vector ordenado dos elementos; a eficiência da pesquisa binária é $O(\log n)$. O "hash" permite uma implementação ainda mais rápida[2], em termos de tempo médio e mesmo (ver 7.3) no pior caso!

– Dicionários dinâmicos: as operações possíveis são a pesquisa, a inserção e, possivelmente, a eliminação.

O vector ordenado é uma má implementação dos dicionários dinâmicos uma vez que a inserção é uma operação ineficiente, $O(n)$. São alternativas melhores: as árvores B e as árvores balanceadas. O método de "hash" é ainda melhor, em termos do tempo médio, uma vez que permite tempos constantes para as 3 operações.

Entre as muitas aplicações do "hashing", referimos a implementação eficiente dos métodos de tabelação ("memoization", ver página 124); estes métodos são usados, por exemplo, em pesquisas heurísticas, e em jogos – por exemplo, para guardar o valor de configurações de um tabuleiro de xadrês.

Formalmente temos

– Um universo U de cardinalidade $u = |U|$. Pode ser, por exemplo, o conjunto de todas as sequências de caracteres com comprimento 100 ($u = 127^{100}$).

– Um conjunto $N \subseteq U$ de elementos a representar na tabela de "hash". A cardinalidade de N é $n = |N|$. O conjunto N poderia ser constituído, por exemplo, pelos nomes dos alunos da Faculdade de Ciências.

– Uma tabela (vector) A de "hash" de cardinalidade $a = |A|$. Normalmente $a \ll u$, tendo a e n valores não muito diferentes, por exemplo, $a = 2n$.

– Define-se uma função de "hash" $h : U \to A$ que a cada valor de U atribui uma posição na tabela A. Para inserir x, coloca-se x na posição $h(x)$ da tabela A, *se essa posição estiver vazia.*

– Quando se insere um valor x e posteriormente um valor y com $h(y) = h(x)$, temos uma *colisão.* A posição da tabela para onde y deveria ir já está ocupada e terá que existir um método para resolver as colisões.

[2]De ordem $O(1)$ se a tabela de "hash" estiver bem dimensionada.

Resolução das colisões

Existem diversos métodos de resolver colisões que se podem classificar globalmente em 2 classes

a) *Métodos internos:* quando há uma colisão o valor a inserir é colocado na própria tabela A no próximo índice livre, sendo a noção de "próximo" dependente do método.

a) *Métodos externos:* cada posição da tabela A contém um apontador para uma lista ligada, inicialmente vazia. Cada valor x a inserir é colocado como primeiro elemento da lista. Neste caso, o haver ou não colisão é irrelevante; a não existência de colisão significa simplesmente que a lista está vazia.

Por hipótese usaremos um método externo.

Propriedades desejáveis da função de "hash"

O que se pretende é que, sendo a não muito maior que n, haja em média poucas colisões. Pretendemos que o método de "hash" tenha as seguintes propriedades:

a) *Espalhamento:* os índices $h(x)$ deverão ficar uniformemente espalhados na tabela, de modo a que haja poucas colisões.

b) *Tabela pequena:* para uma constante pequena k é $a \leq ks$; por outras palavras, a tabela não deverá ser muito maior que o número de elementos a inserir.

c) *Computação rápida:* para todo o elemento x, $h(x)$ deve ser computável em tempo $O(1)$.

Exemplo. Para o caso de um dicionário dos (digamos) 10 000 alunos da Faculdade, poderíamos escolher $a = 20\,000$ e a seguinte função de "hash"

$$((c_1 \ll 12) + (c_2 \ll 8) + (c_{n-1} \ll 4) + c_n) \mod 20\,000$$

onde $n \ll m$ significa que a representação binária de n é deslocada para a esquerda de m bits (o valor correspondente é multiplicado por 2^m).

Infelizmente um simples argumento de contagem, mostra-nos que as propriedades enunciadas podem ser difíceis de obter.

Teorema 21 *Para qualquer função de "hash" h existe um conjunto de $\lceil u/a \rceil$ valores que são mapeados num mesmo índice da tabela A.*

Exercício 79 *Demonstre o Teorema anterior.*

Para o exemplo dado atrás, concluimos que, independentemente da função de "hash" escolhida, existe um conjunto de elementos do universo (isto é, de "strings" de comprimento 100) com cardinalidade $\lceil 127^{200}/20\,000 \rceil > 10^{400}$ que é mapeado num único índice da tabela.

Felizmento veremos que o conhecimento prévio dos elementos a incluir na tabela permite (com um algoritmo aleatorizado) determinar uma função h e uma dimensão da tabela a não muito maior que s tais que não exista qualquer colisão! Este é um assunto que trataremos nas secções 7.2 e 7.3.

Eficiência do método clássico de "hash"

O método de "hash" tradicional (interno e externo) tem sido exaustivamente analisado. Em particular temos no caso médio e supondo que o cálculo de $h(x)$ é efectuado em tempo constante

- Pesquisa, inserção e eliminação: ordem de grandeza $O(1)$. A constante multiplicativa correspondente depende essencialmente do factor $\alpha = n/a$ que mede o grau de preenchimento da tabela.

No pior caso, todas aquelas operações têm tempo de ordem de grandeza $O(n)$, ver o Teorema 21. Ver (muitos) mais pormenores, por exemplo em [6].

7.1.2 Variantes do método de "hash"

Em linhas gerais, temos 3 métodos de "hash":

1) **"Hash" clássico**: a função de "hash" está fixa e os dados têm uma determinada distribuição probabilística. consegue-se

tempo médio $O(1)$

Contudo, no pior caso, (se tivermos muito azar com os dados) os tempos das operações básicas (pesquisa, inserção e eliminação) são de ordem $O(n)$. Supõe-se que o leitor está a par desta análise; breve referência em 7.1.1

2) **"Hash" universal**, aleatorização proveniente da escolha da função de "hash". Consegue-se

tempo médio $O(1)$, *quaisquer que sejamos dados*

Ver 7.2.

3) **"Hash" perfeito**, os valores a memorizar são conhecidos previamente[3]: Consegue-se determinar h por forma que

tempo $O(1)$ mesmo no pior caso

Ver 7.3.

No fundo, estamos a aleatorizar o algoritmo do "hash", da mesma forma que aleatorizamos o algoritmo clássico do "quick sort", ver página 63.

7.2 "Hash" universal: aleatorização do "hash"

Vimos que, qualquer que seja a função de "hash" escolhida, existem conjuntos N de dados que são maus no sentido em que para todo o $x \in N$, $h(x)$ é constante (com uma excepção, todas as listas ligadas de H ficam vazias), ver o Teorema 21. Suponhamos agora que a função de "hash" é escolhida aleatóriamente de um conjunto H de funções; para conjuntos e distribuições probabilísticas convenientes podemos conseguir que, qualquer que seja o conjunto de dados de entrada – não há maus conjuntos de dados –, o valor esperado do tempo de pesquisa seja $O(1)$; estamos obviamente a falar de "valor esperado" relativamente à distribuição probabilística das funções de "hash".

Como já dissemos, o "hash" universal é algo de muito semelhante ao "quick sort" aleatorizado onde, qualquer que seja o vector a ordenar – não há maus vectores – o tempo médio de execução[4] é de ordem $O(n \log n)$.

Definição 11 *Seja H um conjunto de funções de "hash" de U em $\{1, 2, \ldots, a\}$ associada a uma distribuição probabilística a que chamamos também H. Dizemos que H é* universal *se para todo o par de valores $x \neq y$ de U temos*

$$\text{prob}_{h \in_U H}[h(x) = h(y)] \leq 1/a$$

onde $h \in_U H$ significa que a função h é escolhida do conjunto H de forma uniforme, isto é, com igual probabilidade.

Por outras palavras, a probabilidade de haver uma colisão entre 2 quaisquer elementos distintos não excede $1/a$ (praticamente o valor mínimo) onde, lembra-se, a é o tamanho da tabela de "hash".

O seguinte resultado mostra que, relativamente a uma escolha uniforme $h \in_U H$, se esperam poucas colisões entre um qualquer valor x e os outros elementos de um qualquer conjunto N (existente na tabela de "hash"). A grande importância do "hash" universal traduz-se na palavra "qualquer" sublinhada no seguinte teorema.

Teorema 22 *Se H é universal então, para qualquer conjunto $N \subseteq U$ e para qualquer $x \in U$, o valor esperado, relativamente a uma escolha uniforme $h \in_U H$, do número de colisões entre x e*

[4] Relativamente à aleatorização do algoritmo determinada pela escolha dos pivots.

os outros elementos de N não excede n/a (número de elementos de N a dividir pelo tamanho da tabela de "hash").

Dem. Seja $y \in S$ com $x \neq y$. Definimos as seguintes variáveis aleatórias

$$\begin{cases} c_{xy} \text{ com valor 1 se } x \text{ e } y \text{ colidem e 0 caso contrário} \\ c_x \text{ número de colisões com } x\text{: número de elementos } y\text{, diferentes de } x\text{, tais que } h(y) = h(x). \end{cases}$$

Note-se que para o "hash" universal é

$$\begin{aligned} E(c_{xy}) &= 1 \times \text{prob}_{\ h \in_U H}[h(x) = h(y)] + 0 \times \text{prob}_{\ h \in_U H}[h(x) \neq h(y)] \\ &= \text{prob}_{\ h \in_U H}[h(x) = h(y)] \\ &\leq 1/a \end{aligned}$$

Temos $c_x = \sum_{y \in N, y \neq x} c_{xy}$. Usando em primeiro lugar a linearidade do valor esperado e depois a definição de "hash" universal vem

$$E(c_x) = \sum_{y \in N, y \neq x} E(c_{xy}) \leq n/a$$

□

7.2.1 O método matricial de construção

O Teorema 22 traduz a vantagem do "hash" universal: tempo das operações básicas $O(1)$, independentemente dos dados. Mas será que o "hash" universal existe? Será que podemos construir eficientemente a correspondente família de funções de "hash"? Vamos ver que sim.

Escolha aleatória de uma matriz H

Por simplicidade, suponhamos que $a = 2^b$ (a dimensão da tabela de "hash" é uma potência de 2) e que $u = 2^d$ (a dimensão do universo é uma potência de 2). Vamos definir uma matriz H de dimensões $b \times d$; tipicamente H é larga e baixa, por exemplo, com 10 linhas e 1000 colunas. Cada elemento de H é, aleatoriamente e de forma uniforme, 0 ou 1.

Valor de $h(x)$

Seja $x \in U$; x tem d bits. O valor $h(x)$ é

$$h(x) = Hx$$

onde a multiplicação matricial é efectuada "módulo 2" (corpo $\{0, 1\}$).

Exemplo

Suponhamos que é $d = 5$, $b = 3$, $x = [0, 1, 0, 1, 1]^t$ e

$$h(x) = Hx = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Por exemplo, o segundo elemento de $h(x)$ é obtido a partir da segunda linha de H

$$(1 \times 0) + (0 \times 1) + (0 \times 0) + (1 \times 1) + (1 \times 1) = 0 + 0 + 0 + 1 + 1 = 0 \pmod 2$$

Note-se que uma maneira de ver este cálculo é o seguinte: somamos (módulo 2) alguns elementos da segunda linha de H. Quais? Os das linhas i para as quais $x_i = 1$. No exemplo acima, trata-se das linhas 2, 4 e 5 de x; isto é, temos a soma dos elementos de H de cor vermelha, $0 + 1 + 1 = 0 \pmod 2$.

Qual a probabilidade de ser $h(x) = h(y)$?

Consideremos agora dois elementos diferentes, $x, y \in U$, $x \neq y$. Como x e y são diferentes, têm que diferir pelo menos num bit, digamos no bit i; suponhamos então que é, por exemplo, $x_i = 0$ e $y_i = 1$. Vamos atribuir valores a todos os elementos de H, excepto aos da à coluna i. Notemos que, uma vez fixada essa escolha, $h(x)$ é fixo, uma vez que $x_i = 0$ (lado esquerdo da figura seguinte); por outro lado, cada uma das 2^b colunas i possíveis vai originar valores de $h(y)$ diferentes (no lado direito da figura está um desses valores). Para vermos isso, reparemos que, sendo $y_i = 1$, cada alteração do bit da linha j, $(1 \leq j \leq b)$ causa uma alteração do bit j de $h(y)$.

$$\begin{matrix} & & & i & \\ \end{matrix}\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \times \begin{matrix} x \\ \end{matrix}\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{matrix} h(x) \\ \end{matrix}\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \qquad \begin{matrix} & & & i & \\ \end{matrix}\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \times \begin{matrix} y \\ \end{matrix}\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{matrix} h(y) \\ \end{matrix}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Concluimos que, sendo i um bit em que x e y diferem com $x_i = 0$ e $y_i = 1$, as 2^b colunas i possíveis

de H dão origem a 2^b valores de $h(y)$ distintos; assim[5], $h(x)$ e $h(y)$ são variáveis aleatórias independentes sendo $1/2^b = 1/a$ a probabilidade de ser $h(x) = h(y)$. Temos então o seguinte resultado

Teorema 23 *Seja $|A| = a = 2^b$ o tamanho da tabela de "hash" e $|U| = u = 2^d$ o tamanho do universo. Para $x \in U$ seja $h(x)$ definido por $h(x) = Hx$ onde H é uma matriz aleatória e uniforme de 0's e 1's com b linhas e d colunas, e o produto matricial é efectuado no corpo $\{0, 1\}$. Então, este conjunto de funções h é um conjunto universal de "hash".*

7.3 "Hash" perfeito

Com o "hash" universal temos uma garantia de que o *tempo médio* das operações básicas é $O(1)$. Mas o tempo no pior caso pode ser muito mau, nomeadamente $O(n)$. Vamos ver que para dicionários estáticos, utilizando o chamado "hash" perfeito, podemos construir eficientemente uma função de "hash" que garanta tempo $O(1)$, mesmo no pior caso.

Em princípio é desejável que o tamanho a da tabela de "hash" a utilizar não seja muito maior que o número n de elementos que estão lá armazenados, digamos que a deverá ser de ordem $O(n)$. Se permitirmos uma tabela de "hash" de tamanho n^2, a construção (através de um algoritmo aleatorizado) de uma tabela de "hash" perfeito é mais simples, como veremos em 7.3.1.

Se exigirmos uma tabela de "hash" com tamanho linear em n, tal construção também é possível, embora só tal só mais recentemente tenha sido conseguido, ver 7.3.2.

Comecemos por definir formalmente o "hash" perfeito.

Definição 12 *Uma função $h : U \to A$ diz-se uma função de "hash" perfeita para um conjunto dado $N \subseteq U$, se h restrita a N for injectiva (isto é, se não houver colisões entre os elementos de N).*

7.3.1 Construção com espaço $O(n^2)$

Se usarmos uma tabela de "hash" de tamanho n^2 é muito fácil construir uma função de "hash" perfeita usando um algoritmo aleatorizado de tempo polinomial. Com probabilidade pelo menos 1/2 esse algoritmo produz uma função de "hash" perfeita. Como sabemos do estudo dos algoritmos aleatorizados, ver página 65, pode conseguir-se, em tempo da mesma ordem de grandeza, uma probabilidade arbitrariamente próxima de 1.

[5]Relativamente à escolha aleatória dos elementos da coluna i de H.

Basicamente definimos uma função de "hash" aleatória (como no "hash" universal) e, como a tabela é grande, a probabilidade não existir qualquer colisão é grande.

Exercício 80 *São gerados k inteiros aleatórios, de forma unforme, entre 1 e n. Até que valor de k, a probabilidade de não existirem 2 inteiros gerados iguais é superior a 1/2?*

Exercício 81 *Numa sala estão 25 pessoas. Qual a probabilidade de pelo menos 2 dessas pessoas fazerem anos no mesmo dia? Calcule essa probabilidade de forma explícita (e não através de uma expressão). Use os seguintes axiomas: as pessoas nascem com igual probabilidade em todos os dias do ano; nenhum ano é bissexto.*

Teorema 24 *Seja $h(\cdot)$ uma função de "hash" definida através de uma matriz aleatória H correspondente a uma tabela de "hash" com n^2 elementos, construída como se descreveu em 7.2.1. A probabilidade de não haver qualquer colisão é pelo menos 1/2.*

Dem. Existem $\binom{n}{2}$ pares (x, y) de elementos do conjunto a representar. Por definição de "hash" universal, a probabilidade de colisão de um qualquer par específico é não superior a $1/a$. Portanto, a probabilidade p de existir pelo menos uma colisão satisfaz (note-se que $a = n^2$)

$$p \leq \binom{n}{2}/a = \frac{n(n-1)}{2n^2} < \frac{1}{2}$$

□

7.3.2 Construção com espaço $O(n)$

Durante bastante tempo duvidou-se que existisse um algoritmo eficiente de definir uma função de "hash" perfeita que mapeasse um conjunto dado de n elementos numa tabela de "hash" de $O(n)$ elementos. Após diversas tentativas, descobriu-se o seguinte método que é relativamente simples e elegante. Trata-se de um "hash" a 2 níveis. Note-se que o "hash" do primeiro nível não é (excepto se tivermos uma sorte excepcional) perfeito.

Construção do "hash" perfeito

1. Usando o "hash" universal define-se uma função h de "hash" (do primeiro nível) correspondente a uma tabela de "hash" de tamanho n, isto é, $a = n$.

2. Inserem-se todos os n elementos na tabela. Este "hash" não é perfeito, normalmente haverá diversas colisões (cadeias com mais que um elemento) bem como cadeias vazias.

3. Num segundo nível de "hash", é construída para cada cadeia uma função de "hash" perfeito, segundo o algoritmo descrito em 7.3.1. Sejam h_1, h_2,..., h_n as respectivas funções de "hash" e A_1, A_2,..., A_n as respectivas tabelas. Note-se que a tabela A_i tem tamanho n_i^2 em que n_i é o número de elementos da cadeia i, isto é, elementos x com $h(x) = i$.

Antes de mostrar que este esquema aleatorizado produz com probabilidade pelo menos 1/2 um "hash" perfeito, vamos descrever o algoritmo de pesquisa.

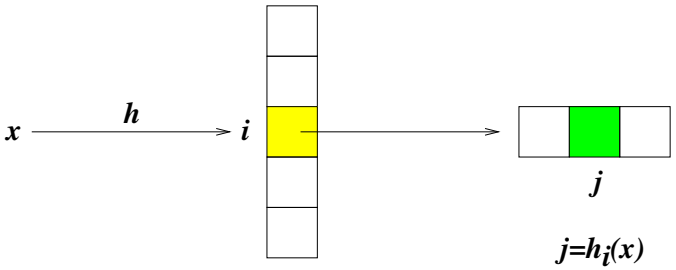
Algoritmo de pesquisa

Dado x, retorna `V` of `F`.

1. Calcula-se $i = h(x)$

2. Calcula-se $j = h_i(x)$

3. Se $A_i[j]$ está ocupado com x, retorna-se `V`, senão (se estiver vazio ou contiver um valor diferente de x) retorna-se `F`



Observação. Vamos mostrar que este algoritmo é $O(1)$.

Observação. As cadeias – elementos com o mesmo valor $h(x)$ – têm muitas vezes 1 ou nenhum elemento; assim, é preferível com vista a poupar espaço e tempo, não construir tabelas de "hash" para esses casos, tratando-os de modo especial.

Correcção do algoritmo

Já verificamos que as funções de "hash" do segundo nível podem ser eficientemente construídas segundo o método de espaço quadrático, 7.3.1. Falta mostrar que o espaço gasto pelas tabelas A_i do segundo nível é $O(n)$.

Lema 2 *Se h é uma função uniformemente escolhida de um conjunto universal,*

$$\text{prob}\left\{\sum_{1\le i\le a} n_i^2 > 4n\right\} < \frac{1}{2}$$

Dem. Basta mostrar que $E(\sum_{1\leq i\leq a} n_i^2) < 2n$; na verdade seja a variável aleatória $Y = \sum_{1\leq i\leq a} n_i^2$. Pela desigualdade de Markov se $E(y) < 2n$, então prob $(Y > 4n) < 1/2$.

Se contarmos o número total de pares que colidem no "hash" do primeiro nível, incluindo as colisões dos elementos consigo próprios, vamos ter $\sum_{1\leq i\leq a} n_i^2$. Por exemplo, se a, b e c são os elementos de uma das cadeias, temos $3^2 = 9$ colisões que correspondem a todos os pares (x, y) com x e y pertencentes a $\{a, b, c\}$. Assim, e definindo-se c_{xy} como sendo 1 se x e y colidem e 0 caso contrário, temos que o valor médio da soma dos quadrados é

$$\begin{array}{rcll} E(\sum_i n_i^2) & = & \sum_x \sum_y E(c_{xy}) & \\ & = & n + \sum_x \sum_{y\neq x} E(c_{xy}) & \\ & \leq & n + n(n-1)/a & \text{porque } h \text{ é universal} \\ & \leq & n + n(n-1)/n & \text{porque } a = n \\ & < & 2n & \end{array}$$

□

7.4 Contar o número de elementos distintos

Consideremos o problema de determinar de forma eficiente quantos elementos distintos[6] existem numa longa sequência de, por exemplo, "strings".

Uma solução é usar um método de "hash", não inserindo um elemento quando ele já se encontra na tabela. No fim, conta-se quantos elementos existem na tabela.

Mas suponhamos agora que a sequência é realmente muito grande e que não dispomos de espaço suficiente para ter uma tabela de "hash". Por outro lado, aceitamos obter apenas um valor aproximado do número de elementos distintos. Por exemplo, podemos observar um "router" durante uma hora e pretendemos saber aproximadamente quantos IP's distintos existiram nas mensagens que foram encaminhadas pelo "router" nesse intervalo de tempo.

Uma solução é gerar uma função h de "hash" que produza um resultado uniforme em $[0, 1]$ (intervalo real) e determinar $\min_x h(x)$; se colocarmos de forma aleatória e uniforme p pontos $h(x_i)$ em $[0, 1]$, o valor médio de $\min_x h(x)$ é $1/(p + 1)$. Assim, calculamos esse mínimo e, a partir dele, o valor aproxmado de p. Obviamente, aqui não existe tabela de "hash", mas o método é inspirado nas funções de "hash".

Referências. Os leitores interessados na relação entre o hash universal, a criptografia e a "desaleatorização" de algoritmos podem consultar por exemplo "Pairwise Independence and Derandomization" de Michael Luby e Avi Wigderson, acessível de

`http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/CSD-95-880.pdf`

[6] Se não houvesse a imposição de os elementos serem distintos, a solução era trivial...

ou "Applications of Universal Hashing in Complexity Theory" de V. Arvind e M. Mahajan,

wwwcs.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts_from_www/hash-survey.ps.gz

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*	$\underline{4}$

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	$\underline{4}$

[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.

Capítulo 9

Sobre o algoritmo FFT

9.1 Transformações de representação, generalidades

9.2 Polinómios em corpos. Representações

9.2.1 Cálculo de um polinómio num ponto

Consideremos um polinómio de grau $n-1$

$$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \tag{9.1}$$

representado pelos seus n coeficientes a_0, a_1,..., a_{n-1}. Pretendemos calcular $A(x)$ para um valor particular x. O método directo baseado em (9.1) envolve

- $n-2$ multiplicações para calcular os valores de x^2, x^3,..., x^{n-1}
- $n-1$ multiplicações para calcular os valores de $a_0 x$, $a_1 x$,..., $a_{n-1} x^{n-1}$.

num total de $2n-1$ multiplicações

Regra de Horner

Um método mais eficiente de calcular $A(x)$ é baseado na seguinte igualdade

$$A(x) = a_0 + x(a_1 + x(a_2 + \ldots + x a_{n-1} \ldots)) \tag{9.2}$$

Por exemplo,

$$2 + 3x + 4x^2 + 5x^3 = 2 + x(5 + x(4 + x \times 5))$$

Exercício 86 *Mostre a validade da regra de Horner. Sugestão: use indução em n.*

Exercício 87 *Escreva numa linguagem da sua preferência (C, python, haskell...) uma função que tem como argumentos um valor x e o vector dos coeficientes de um polinómio e calcula o valor desse polinómio no ponto x, usando a regra de Horner.*

Exercício 88 *Mostre que o número de multiplicações que são efectuadas para se calcular o valor de $A(x)$ usando a regra (9.2.1) é $n-1$, isto é, igual ao grau do polinómio.*

9.2.2 Dois modos de representar um polinómio

Dois modos de representar um polinómio $A(x)$ de grau $n-1$ são

(C) A sequência dos seus coeficientes $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. O polinómio é $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$.

(V) A sequência de n valores que o polinómio toma em n pontos pré-estabelecidos, x_0, x_1,..., x_{n-1}:

$$\langle y_0, y_1, \ldots, y_{n-1}\rangle$$

É sabido que, fixados os valores x_0, y_0, x_1, y_1,..., x_{n-1} e y_{n-1} (sendo todos os x_i distintos) existe um e um só polinómio $A(x)$ tal que $A(x_0) = y_0$, $A(x_1) = y_1$, ... e $A(x_{n-1}) = y_{n-1}$. A fórmula de interpolação de Lagrange permite "reaver" os coeficientes do polinómio

$$A(x) = \sum_{j=0}^{n-1} \left(\prod_{k=0, k\neq j}^{n-1} \frac{x - x_k}{x_j - x_k} y_j \right)$$

Esta fórmula é fácil de justificar. Note-se que para caracterizar univocamente um polinómio de grau $n-1$ necessitamos de n valores y_0, y_1,..., e y_{n-1}.

Nota importante. A escolha dos n pontos x_0, x_1,..., e x_{n-1} é perfeitamente arbitrária; apenas se impõe que sejam distintos. Esta liberdade de escolha permitirá a implementação de um algoritmo muito eficiente (FFT) para efectuar a conversão de uma representação para outra.

Vejamos um exemplo.

Exemplo. Suponhamos que um polinómio é definido pelos seus valores em 3 pontos

x_i	y_i
0	-1
1	0
3	1

Isto é, $x_0 = 0$, $x_1 = 0$, $x_2 = 3$, $y_0 = -1$, $y_1 = 0$, $y_2 = 1$. Apliquemos a fórmula de Lagrange

$$A(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2$$

É fácil ver-se que $A(x_0) = y_0$, $A(x_1) = y_1$ e $A(x_2) = y_3$. Temos

$$A(x) = \frac{(x-1)(x-3)}{(-1)(-3)}(-1) + \frac{x(x-3)}{1(-2)}0 + \frac{x(x-1)}{3\times 2}1 = -\frac{(x-1)(x-3)}{3} + \frac{x(x-1)}{6}1$$

ou seja, $A(x) = -x^2/6 + (7/6)x - 1$. Pode facilmente verificar-se que $A(0) = -1$, $A(1) = 0$ e $A(2) = 1$. □

9.2.3 Multiplicação de 2 polinómios

Suponhamos que pretendemos multiplicar 2 polinómios

$$\begin{aligned} A(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \\ B(x) &= b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1} \end{aligned}$$

O cálculo deste produto no domínio dos coeficientes (C) é relativamente trabalhoso; há que efectuar o produto

$$A(x)B(x) = (a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1})(b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1})$$

calculando o coeficiente de cada x^i, para $i = 0, 1, 2\ldots, n-2$.

Multiplicação no domínio dos valores (C); uma aplicação da "transformação de representações"

Suponhamos que calculamos $A(x)$ e $B(x)$ para $2n-1$ valores distintos e pré-fixados x_0, x_1,..., e x_{n-1}. Seja $y_i = A(x_i)$ e $y'_i = B(x_i)$ para $0 \le i \le 2n-1$. Estamos agora no domínio dos

valores, (V). Neste domínio, como se representa o produto $A(x)B(x)$? Muito fácil, temos apenas que multiplicar os valores um a um

$$A(x)B(x) \leftrightarrow \langle y_0y'_0, y_1y'_1, \cdots, y_{2n-1}y'_{2n-1}\rangle$$

Efectuamos apenas $2n-1$ multiplicações de inteiros; tivemos de calcular $A(x)$ e $B(x)$ em $2n-1$ pontos porque o produto $A(x)B(x)$ tem grau $2n-2$.

Em resumo,

1. $A(x)$, (C)→(V), converter $\langle a_0, a_1, \ldots, a_{n-1}\rangle \rightarrow \langle y_0, y_1, \ldots, y_{2n-2}\rangle$
2. $B(x)$, (C)→(V), converter $\langle b_0, b_1, \ldots, b_{n-1}\rangle \rightarrow \langle y'_0, y'_1, \ldots, y'_{2n-2}\rangle$
3. Efectuar $2n-1$ produtos (de números) para obter $\langle y_0y'_0, y_1y'_1, \ldots, y_{2n-1}y'_{2n-1}\rangle$
4. $A(x)B(x)$, (V)→(C), converter $\langle y_0y'_0, y_1y'_1, \ldots, y_{2n-1}y'_{2n-1}\rangle \rightarrow \langle c_0, c_1, \ldots, c_{2n-1}\rangle$

Este só se poderá tornar um método eficiente de multiplicar polinómios se conseguirmos algoritmos de conversão eficientes; esses algoritmos eficientes existem, são o FFT, tanto para a conversão (C)→(V) como para a conversão (V)→(C).

9.2.4 Corpos

Definição 14 *Um corpo $(\mathcal{U}, +, \times)$ é uma estrutura algébrica definida num domínio $\mathcal{U}$, com as operações "+" e "×" com as seguintes propriedades*

- *$(\mathcal{U}, +)$ é um grupo abeliano; o elemento neutro designa-se 0, o inverso aditivo de a designa-se $(-a)$.*
- *$(\mathcal{U} \setminus \{0\}, \times)$ é um grupo abeliano: o elemento neutro designa-se 1, o inverso multiplicativo de a designa-se a^{-1}.*
- *Para quaisquer elementos a, b, $c \in \{\mathcal{U}\}$ é $a(b+c) = ab + ac$ e $(a+b)c = ac + bc$.*

Nota. Omitiremos frequentemente o sinal "×", escrevendo ab em vez de $a \times b$.

Nota. Para $m \in \mathbb{N}$ representamos por a^m o produto de m termos $aa\cdots a$; em particular $a^0 = 1$ e $a^1 = a$. Note-se que a potenciação não é uma operação associada aos corpos, mas apenas uma convenção de escrita,

$$a^m \equiv \overbrace{a \times a \times \cdots \times a}^{m\ a\text{'s}}$$

Exemplos. O conjunto dos números reais $\mathbb{R}$ com a adição e a multiplicação usuais é um corpo. O conjunto dos números complexos com a adição e a multiplicação usuais é um corpo.

Exemplo. O anel das classes residuais módulo n (com as operações de adição e a multiplicação módulo n) é um corpo sse n é primo. Vamos designá-lo por $\mathbb{Z}_p$.

Exercício 89 *Verifique que $\mathbb{Z}_5$ é um corpo.*

9.2.5 Raízes primitivas da unidade

Os n valores x_0, x_1, x_{n-1} para os quais vamos calcular um polinómio estão directamente relacionados com as raízes primitivas da unidade. Seja $(\mathcal{U}, +, \times)$ um corpo[1].

Definição 15 *Seja $(\mathcal{U}, +, \times)$ um corpo. Uma raiz primitiva da unidade de ordem n, designada por ω_n, é um elemento de $\mathcal{U} \setminus \{0\}$ tal que*

(1) ${\omega_n}^n = 1$

(2) Para qualquer m com $1 \leq m < m$ é ${\omega_n}^m \neq 1$

Nota. Para $i \in \mathbb{N}$ representamos ${\omega_n}^i = 1$ por $\omega_n^i = 1$; entende-se que é o produto de i factores todos iguais a ω_n.

Nota. A raíz de ordem 0 da unidade é 1, a própria unidade.

Exercício 90 *Mostre que no corpo $(\mathbb{R}, +, \times)$ dos reais as únicas raízes primitivas da unidade são 1 e -1.*

Teorema 28 *Seja p um primo. Então $(\mathbb{Z}_p, +, \times)$ onde $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$, "$+$" é a adição módulo p e "$\times$" é a multiplicação módulo p é um corpo.*

Exercício 91 *Mostre que $\{0, 1, 2, 3, 3\}$ com a adição módulo 4 e a multiplicação módulo 4 não formam um corpo.*

[1] Na realidade, para esta secção basta considerar o grupo $(\mathcal{U}, \times)$

Exercício 92 *Considere o corpo* $(\mathbb{Z}_7, +, \times)$. *Averigue quais dos elementos 1, 2, 3, 4, 5, 6 e 7 são raízes primitivas da unidade e determine a respectiva ordem.*

Exercício 93 *Considere o corpo dos números complexos* $(\mathbb{C}, +, \times)$. *Sejam* i *e* m *inteiros positivos, primos entre si. Mostre que o número* $e^{2\pi i/m}$ *é uma raíz primitiva da unidade de ordem* m. *Use a interpretação geométrica do complexo* $e^{\alpha i}$ *com* $\alpha \in \mathbb{R}$.

9.3 A DFT: dos coeficientes para os valores

Vamos calcular o valor de um polinómio de grau não superior a $n-1$ em n pontos x_0, x_1,..., x_{n-1}. Seja o polinómio $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$. Esse cálculo corresponde à seguinte multiplicação de matrizes

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \cdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & \cdots & x_0^{n-1} \\ x_1^0 & x_1^1 & \cdots & x_1^{n-1} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n-1}^0 & x_{n-1}^1 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{bmatrix}$$

Nota. Insistimos: na equação em cima x_i^j designa o valor x_i levantado à potência j.

9.3.1 O que é a transformada discreta de Fourier, DFT?

Definição 16 *Seja* ω *uma raíz primitiva de ordem* n *da unidade. Consideremos um polinómio* $A(x)$ *representado pelos coeficientes* $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. *A* transformada discreta de Fourier *de* $A(x)$ *é o conjunto de valores*

$$\langle A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})\rangle$$

Nota. Por definição de "raíz primitiva da unidade", os valores ω^0, ω^1, ..., ω^{n-1} são todos distintos.

Exercício 94 *Verifique a veracidade da nota anterior.*

Nota. O cálculo de $A(x)$ nos pontos referidos pode ser traduzido na seguinte equação matricial

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \cdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{bmatrix} \tag{9.3}$$

ou, de forma mais compacta

$$\hat{A} = \Omega_n A$$

Exercício 95 *Verifique a veracidade da nota anterior.*

Nota. A escolha criteriosa dos pontos em que A é avaliado, nomeadamente

$$x_0 = 1,\ x_1 = \omega,\ x_2 = \omega^2,\ \ldots,\ x_{n-1} = \omega^{n-1}$$

permitirá uma implementação muito eficiente da computação $\hat{A} = \Omega_n A$: o algoritmo FFT.

9.3.2 A inversa da transformada discreta de Fourier

A matriz Ω_n, ver (9.3) (página 143), sendo uma matriz de Van der Monde, não é singular. A sua inversa obtém-se substituindo ω por ω^{-1} e multiplicando pelo escalar $1/n$, isto é

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)^2} \end{bmatrix}^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega{-}1 & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \ldots & \omega^{-(n-1)^2} \end{bmatrix} \tag{9.4}$$

Antes de demonstrarmos esta igualdade no caso geral, vamos considerar Ω_3 no corpo dos complexos com a raíz primitiva da unidade $\omega = e^{1\pi i/n}$. Algumas igualdades que utilizaremos em seguida (para o caso $n = 3$) são

$$\omega^3 = 1, \quad \omega^{-1} = \omega^2, \quad \omega^{-2} = \omega, \quad 1 + \omega + \omega^2 = 0$$

(para verificar a última igualdade, considerar a representação vectorial de $e^0 = 1$, $e^{2\pi i/n} = \omega$ e $e^{4\pi j/n} = \omega^2$).

Façamos então o produto

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega^4 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega^{-1} & \omega^{-2} \\ 1 & \omega^2 & \omega^{-4} \end{bmatrix}$$

ou seja

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega^2 & \omega \\ 1 & \omega & \omega^2 \end{bmatrix}$$

Obtemos

$$\begin{bmatrix} 3 & 1+\omega+\omega^2 & 1+\omega^2+\omega \\ 1+\omega+\omega^2 & 1+\omega^3+\omega^3 & 1+\omega^2+\omega \\ 1+\omega^2+\omega & 1+\omega+\omega^2 & 1+\omega^3+\omega^3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

como deveria ser!

Teorema 29 *Seja $A(x)$ um polinómio definido num corpo arbitrário com grau $n-1$ e seja ω uma raíz primitiva da unidade de grau n. Para transformarmos o vector A dos n coeficientes de $A(x)$ no vector $\hat{A}$ dos n valores de $A(x)$ em ω^0, ω,...ω^{n-1}, efectuamos a operação matricial $\hat{A} = \Omega_n A$. Para transformarmos o vector $\hat{A}$ dos n valores de $A(x)$ em ω^0, ω,...ω^{n-1} n no vector A dos coeficientes de $A(x)$, efectuamos a operação matricial $A = \Omega_n^{-1}\hat{A}$.*
A matriz Ω_n^{-1} obtém-se da matriz Ω_n substituindo ω por ω^{-1} e multiplicando pelo escalar $1/n$.

Dem. Designemos por M a matriz que se obtém da matriz Ω_n substituindo ω por ω^{-1} e multiplicando pelo escalar $1/n$, ver (9.4), página 143. Pretende-se mostrar que $M = \Omega^{-1}$. O elemento da linha i, coluna k de Ω é ω^{ik} e o elemento da linha k, coluna j de M é ω^{-kj}. Assim, o elemento da linha i, coluna j de ΩM é

$$r_{i,j} = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj}$$

Se $i = j$, $r_{i,j} = 1$, pois trata-se de uma soma de n termos, todos iguais a 1 multiplicada por $(1/n)$. Se $i \neq j$, seja $m = i - j$; a soma anterior pode escrever-se

$$r_{i,j} = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{mk} = \frac{1}{n} \times \frac{(\omega^m)^n - 1}{\omega^m - 1}$$

pois trata-se de uma soma geométrica de razão ω^m. Mas $(\omega^m)^n = (\omega^n)^m = 1^m = 1$. Logo,

para $i \neq j$, temos $r_{i,j} = 0$. □

Nota. O Teorema anterior é válido num qualquer corpo.

Nota. O Teorema anterior mostra que a transformação DFT e a transformação inversa são muito semelhantes; um bom algoritmo para uma dessas transformações adapta-se facilmente à outra. Tal algoritmo é o FFT, estudado na próxima secção.

9.4 O algoritmo FFT

O algoritmo FFT é uma implementação eficiente da DFT, transformada discreta de Fourier, ver (9.3), página 143.

Suponhamos que n é par, seja[2] $n = 2^m$. O polinómio $A(x)$ pode ser decomposto nos termos de grau par e de grau ímpar, como se ilustra para (termos pares a azul, ímpares a vermelho)

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1} \\ &= (a_0 + a_2 x^2 + \ldots + a_{n-2} x^{n-2}) + (a_1 x + a_3 x^3 + \ldots + a_{n-1} x^{n-1}) \\ &= (a_0 + a_2 x^2 + \ldots + a_{n-2} x^{n-2}) + x(a_1 + a_3 x^2 + \ldots + a_{n-1} x^{n-2}) \\ &= A_P(x^2) + x A_I(x^2) \end{aligned}$$

onde $A_P(x)$ e $A_I(x)$ têm os coeficientes de grau par e ímpar, respectivamente

$$\begin{aligned} A_P(x) &= \sum_{i=0}^{m-1} a_{2i} x^i \\ A_I(x) &= \sum_{i=0}^{m-1} a_{2i+1} x^i \end{aligned}$$

Usando a decomposição $A(x) = A_P(x^2) + x A_I(x^2)$ vamos reescrever o produto; indicamos em A

[2] Mais tarde, para aplicar o esquema "dividir para conquistar", vamos supor que $n = 2^k$, isto é, que as divisões $n \to n/2 \to n/4 \to \cdots \to 1$ são sempre exactas

(coeficientes) e em $\hat{A}$ (valores) o número de termos em índice:

$$\hat{A}_n(x) = \hat{A}_{2m}(x) = \Omega_{2m} A_{2m} \quad = \quad \begin{bmatrix} A(1) \\ A(\omega) \\ \dots \\ A(\omega^{m-1}) \\ A(\omega^m) \\ A(\omega^{m+1}) \\ \dots \\ A(\omega^{2m-1}) \end{bmatrix} = \begin{bmatrix} A_P(1) + A_I(1) \\ A_P(\omega^2) + \omega A_I(\omega^2) \\ \dots \\ A_P((\omega^{m-1})^2) + \omega^{m+1} A_I((\omega^{m-1})^2) \\ A_P(1) + \omega^m A_I(1) \\ A_P(\omega^2) + \omega^{m+1} A_I(\omega^2) \\ \dots \\ A_P((\omega^2)^{m-1}) + \omega^{2m-1} A_I((\omega^2)^{m-1}) \end{bmatrix}$$

A equação anterior permite-nos esquematizar o algoritmo FFT

Dados: vector dos coeficientes, número n de coeficientes e ω, raiz primitiva da unidade de ordem n.

Resultado: vector V, transformada discreta de Fourier de $[a_0, a_1, \dots, a_{n-1}]$

Nota: Se $n \geq 2$ há 2 chamadas recursivas a vectores com metade do tamanho

FFT($[a_0, a_1, \dots, a_{n-1}]$,n,$\omega$):
  `if` $n = 1$, `return` a_0
  `if` $n \geq 2$, seja $m = n/2$
    $\mathcal{P}$ = FFT($[a_0, a_2, \dots, a_{n-2}]$,m,$\omega^2$) (calculo recursivo)
    $\mathcal{I}$ = FFT($[a_1, a_3, \dots, a_{n-1}]$,m,$\omega^2$) (calculo recursivo)
    `for` $i = 0, 1, \dots, m-1$:
      $V[i] = P[i] + \omega^i I[i]$
      $V[m+i] = P[i] + \omega^{m+i} I[i]$
    `return` V

9.4.1 Análise da eficiência do algoritmo FFT

Dado ω, uma raíz primitiva de ordem n da unidade, as potências ω^2, ω^3,..., ω^{n-1} são pré-calculadas. Este cálculo é efectuado em tempo $O(n)$.

Do algoritmo FFT pode ver-se que, se $n = 1$ o tempo de execução é $O(1)$ e que, se $n \geq 2$, o tempo total $t(n)$ é a soma de 2 parcelas

- uma parcela $O(n)$ (ciclo `for`)
- $2t(n/2)$ (correspondente às 2 chamadas recursivas)

Assim, a seguinte recorrência fornece um majorante para $t(n)$

$$\begin{cases} t(1) = a \\ t(n) = 2t(n/2) + an \quad \text{se } n \geq 2 \end{cases}$$

Nota. Como estamos interessados apenas num majorante do tempo de execução, usamos o mesmo coeficiente a como majorante de $t(1)$ e como coeficiente de n.

Exercício 96 *Mostre que a solução da recorrência anterior é* $t(n) = n(1 + \log n)a$.

Fica assim provado que (usando o modelo uniforme na análise da complexidade) a DFT de um vector com n elementos pode ser calculada em tempo $O(n \log n)$.

Esta ordem de grandeza é muito melhor (tempo assintótico muito menor) que a que corresponde à utilização da regra de Horner (ver página 138) para calcular um polinómio de grau n em n pontos: n^2 multiplicações. Por exemplo, fazendo $a = 1$ e supondo que a unidade do tempo é 1 μs (1 microsegundo), temos os seguintes tempos de execução de uma DFT com $n = 65\,536 = 2^{16}$:

Método	número de operações	tempo (seg)
R. Horner	$2^{16} \times 2^{16} = 2^{32} = 4\,294\,967\,296$	4300
FFT	$2^{16} \times 17 = 1\,114\,112$	1.11

Sublinha-se: com as hipóteses feitas, o FFT demoraria cerca de 1 segundo e a regra de Horner mais que 1 hora.

9.5 Aplicações

Vamos apenas dar uma vaga ideia da grande aplicabilidade prática da FFT.

Em Matemática, em Física e em Engenharia utilizam-se transformações de funções que facilitam o o seu tratamento ou estudo. Duas dessas transformações são a transformada de Laplace e a transformada de Fourier. Certas operações são mais eficientes com funções transformadas.

Por exemplo, a transformada de Fourier de uma função $g\colon \mathbb{R} \to \mathbb{C}$ é uma função $h\colon \mathbb{R} \to \mathbb{C}$. Se $g(t)$, a função original, for um sinal sonoro, o valor $g(t)$ é evidentemente um valor real. A transformada de Fourier $h(f)$ (onde o parametro f representa a frequência) é uma função com variável real (a frequência) e valor complexo que tem informação "espectral" de $f(t)$. A transformada de Fourier permite-nos passar do *domínio dos tempos* para o *domínio dos frequências*.

Por exemplo, se $g(t)$ for "próxima" de uma sinusóide de frequência f', então $h(f)$ vai ter um "pico" para $f \approx f'$, ver um exemplo nas figuras 9.5 e 9.5.

Mais precisamente, se $h(f) = a + bi$, a "amplitude" ("conteúdo" do sinal na frequência f) é $|a + bi| = \sqrt{a^2 + b^2}$ e a fase é $\text{arc}(a, b)$ onde $\text{arc}(a, b)$ representa o ângulo, compreendido entre 0

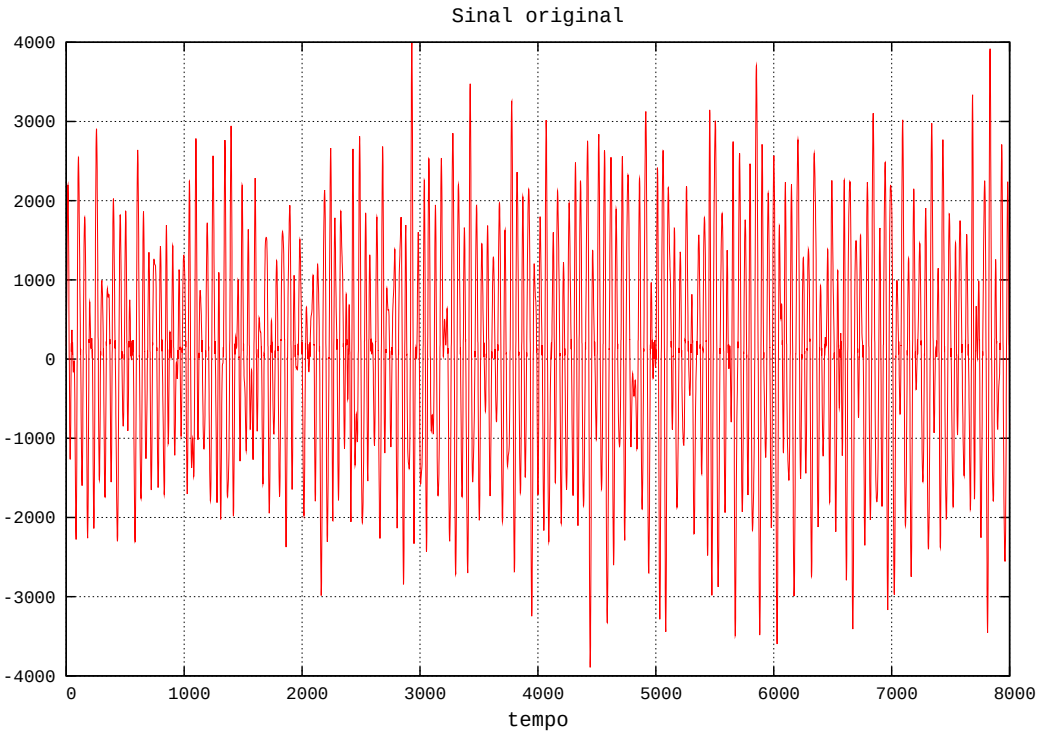


Figura 9.1: Sinal sonoro original (assobio); obtido com a adaptação dos programas de `www.captain.at/howto-fftw-spectrograph.php` baseados em bibliotecas existentes em `www.fftw.org`

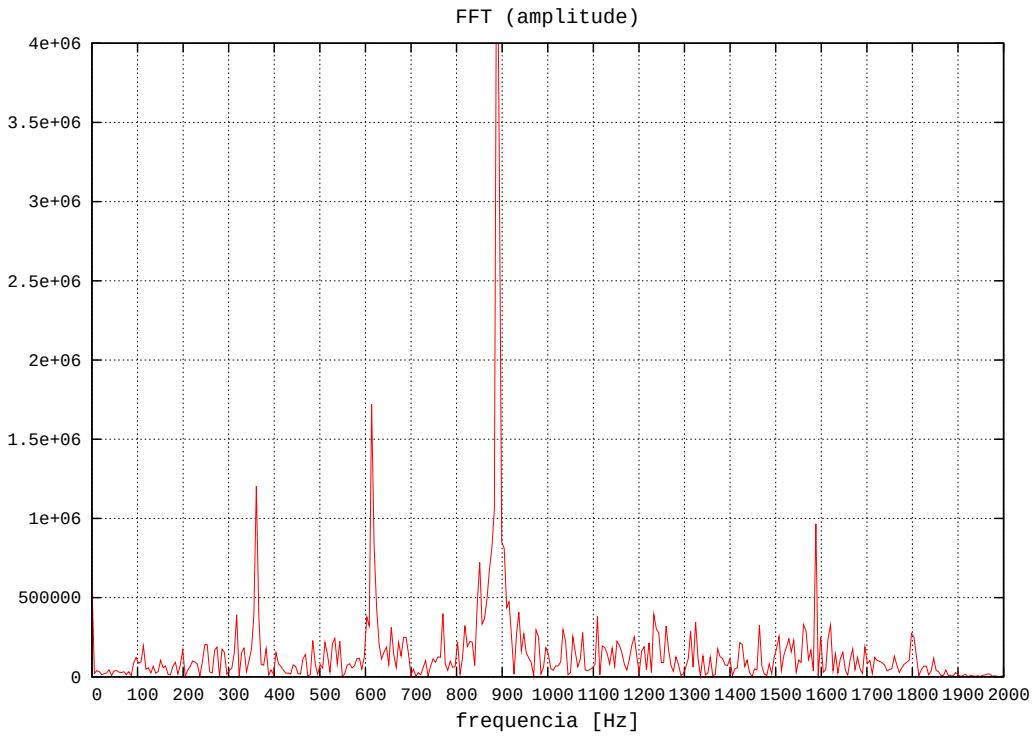


Figura 9.2: Amplitude da transformada discreta de Fourier do sinal da Figura 9.5, obtida com o algoritmo FFT; a fase não é representada. Note-se a predominância de uma frequência de cerca de 900 Hz.

e 2π, entre o eixo dos XX e o vector de coordenadas (x, y). Do ponto de vista da percepção humana, a amplitude (representada na figura 9.5) é muito mais importante que a fase.

Exemplo. No método de codificação mp3 efectua-se uma compressão com perdas do sinal sonoro. Isto é, $f(t)$ é simplificado por forma a que o sinal comprimido ocupe menos espaço. Não é possível reconstituir o sinal original $g(t)$ a partir do sinal simplificado. Em que consiste "simplificar" um sinal? Uma das operações de simplificação é a seguinte

1. Calcula-se $\text{FFT}(g(t)) \equiv h(f)$.
2. Se $h(f)$ contiver "picos" de amplitude fortes com "picos" próximos fracos, eliminam-se esses "picos" fracos, obtendo-se $h'(f)$.
3. Calcula-se $\text{FFT}^{-1}(h'(t)) \equiv f'(t)$.
4. Comprime-se $f'(t)$

O princípio usado baseia-se no seguinte facto experimental: se um sinal tiver 2 "picos" em frequências próximas, um muito mais forte que o outro, a parte do sinal correspondente ao "pico" mais baixo pode ser quase imperceptível (dependendo da próximidade das frequências e da disparidade das amplitudes). Os puristas audiófilos afirmam que se perde sempre qualidade ao codificar um sinal sonoro em mp3 e que é preferível usar sistemas de codificação sem perdas, como o flac. (fim do exemplo)

A análise espectral pode também ser usada noutras situações (por exemplo na análise no processamento das imagens em medicina) e tem uma vasta aplicabilidade prática.

Em resumo

- A transformada de Fourier e a transformada de Laplace são transformações de funções contínuas extremamente utilizadas. A análise e o processamento dos sinais são muitas vezes mais facilmente efectuados no domínio das transformadas.
- A transformada discreta de Fourier (DFT) é de certo modo uma aproximação da transformada de Fourier; o argumento das funções é discreto e não contínuo.
- A algoritmo FFT, formalizado por Cooley e Tukey (mas provavelmente já conhecido por Gauss) permite calcular eficientemente a DFT.

9.5.1 Multiplicação eficiente de matrizes

9.5.2 Transformadas tempo ↔ frequência

- Corpo dos complexos, no sinal original a parte imaginária é 0.

- Na transformada $a+bi$, a amplitude $\sqrt{a^2+b^2}$ é a parte mais importante; a fase é $\arctan2(a,b)$.
- MP3

Capítulo 10

Notas sobre minorantes de complexidade

Neste capítulo faremos uma introdução às aplicações da teoria da Informação aos problemas de complexidade mínima, isto é, à procura de *minorantes* da complexidade da classe dos algoritmos que resolvem um determinado problema.

Antes de definir o que é um minorante ou um majorante de complexidade de um problema temos que convencionar o modelo de custos adoptado e a forma de o medir. Por exemplo

- Modelo de custos: tempo, espaço, número de comparações...
- Medição: pior caso, caso médio...

Definição 17 *Consideremos um problema P e seja n o comprimento dos dados. Diz-se que $f(n)$ é um* majorante da complexidade *de P para um certo modelo de custos $t(n)$ se existe um algoritmo A que resolve P com um custo de execução $t(n)$ que satisfaz $t(n) \leq f(n)$. Formalmente*

$$\exists\ A, \forall n :\ t(A(n)) \leq f(n)$$

onde, por exemplo no modelo do tempo no pior caso, é $t = \max_{|x|=n} t(A(x))$.

Note-se que para mostrar que $f(n)$ é majorante da complexidade de P basta encontrar um algoritmo específico que resolve P com um custo, no pior caso, que não excede $f(n)$.

Definição 18 *Consideremos um problema P e seja n o comprimento dos dados. Diz-se que $g(n)$ é um* minorante da complexidade *de P para um certo modelo de custos $t(n)$ se qualquer algoritmo*

que resolve P tem um custo $t(n)$ que satisfaz $t(n) \geq f(n)$ para uma infinidade de valores de n. Formalmente

$$\forall\ A,\ \exists_\infty n, \exists x,\ |x| = n :\ t(A(x)) \geq f(n)$$

□

Seja P um problema dado e seja $f(n)$ a complexidade (por exemplo, em termos de ordem de grandeza) do algoritmo mais eficiente A que resolve esse problema (eventualmente pode haver mais do que um); chamemos a $f(n)$ a complexidade óptima do problema. Normalmente o algoritmo óptimo A é desconhecido mas, tanto do ponto de vista teórico como do prático, é importante conhecer o mais exactamente possível — isto é, limitar a um intervalo tão pequeno quanto possível — a complexidade óptima $f(n)$.

A complexidade de qualquer algoritmo que solucione P é obviamente um majorante da complexidade óptima.

Determinar minorantes de $f(n)$ é muito mais difícil uma vez que, para se mostrar que $g(n)$ é um minorante de $f(n)$ é necessário considerar *todos* os algoritmos para P e mostrar que nenhum deles tem uma eficiência melhor do que $f(n)$; como o número desses algoritmos é infinito, um ataque directo à determinação de minorantes está fora de questão.

A Teoria da Informação pode ser utilizada para a determinação de minorantes de complexidade. A ideia básica, em linhas gerais é a seguinte. Para que um algoritmo consiga solucionar todas as "instâncias" de um determinado comprimento é necessário que obtenha, da "instância" em questão, informação suficiente para poder determinar a solução correcta. Por outras palavras, o facto de se "seleccionar" deterministicamente, de entre todas as soluções possíveis, a solução correcta, obriga a uma obtenção de informação da "instância" que representa, por si só, uma tarefa que qualquer algoritmo que resolva o problema dado tem que executar.

Neste trabalho discutiremos primeiro problemas em que a informação é obtida por pesagens em balanças de pratos. Seguidamente consideraremos a comparação entre valores numéricos como fonte de informação em problemas como a determinação de máximo, o "merge" de 2 sequências e a ordenação.

10.1 Exemplos introdutórios

10.1.1 Um problema simples

Existem 3 bolas, sendo 2 de peso igual e uma mais pesada; podemos fazer apenas uma "pesagem" numa balança de pratos que nos dá 3 resultados possíveis: ou cai para a esquerda ou para a direita

ou fica equilibrada. Qual a bola mais pesada? A resposta é simples: comparemos com a balança o peso da bola 1 com a bola 2:

- Se forem iguais, a bola 3 é a mais pesada de todas.
- Se a bola 1 pesa mais que a 2, então a bola 1 é a mais pesada de todas.
- Se a bola 2 pesa mais que a 1, então a bola 2 é a mais pesada de todas.

Neste caso, o número de soluções que o problema pode ter é 3 (qualquer uma das bolas pode ser a mais pesada) e o número de resultados possíveis das pesagens é também 3. Em geral podemos enunciar o seguinte Teorema.

Teorema 30 (Princípio da informação necessária) *Para que seja sempre possível chegar à solução é condição necessária que $a \leq b$ onde a representa o número de soluções do problema e b representa o número de casos que é possível distinguir com a informação obtida pelo algoritmo.*

Se são permitidas k pesagens, o número de resultados possíveis das pesagens é 3^k.

Problema

Existem 14 bolas, todas iguais, com excepção de uma que pode ser mais ou menos pesada que as outras; com 3 pesagens determinar qual a diferente e se é mais leve ou pesada que as outras.

Neste caso $a = 28$ pois qualquer uma das 14 bolas pode ser a mais leve ou a mais pesada e $b = 3^3 = 27$; como $a > b$, podemos afirmar que não existe solução!

Problema

Existem 5 bolas todas de pesos diferentes; determinar, com um máximo de 4 pesagens, a sequência das bolas ordenadas pelos seus pesos.

Temos $a = 5! = 120$, (todas as permutações das 5 bolas) e $b = 3^4 = 81$; mais uma vez o problema não tem solução. Qual o número mínimo de pesagens para que o problema *possa* ter solução? Admitamos que os pesos são tais que a balança nunca fica equilibrada (o que é consistente com o facto de os pesos serem todos diferentes). Cada pesagem só dá origem a 2 decisões possíveis. Assim temos que p, o número de pesagens, deve satisfazer $2^p \geq 120$, ou seja

$$p \geq \lceil \log 120 \rceil = 7$$

10.1.2 O problema das 12 bolas

O Teorema 30 anterior não só é útil para determinar casos de impossibilidade mas pode ser também usado para seleccionar as pesagens que podem constituir uma solução. Vejamos um problema com solução e procuremo-la.

> **Problema**
>
> Existem 12 bolas, todas iguais, excepto uma (que pode ser mais ou menos pesada que as outras); com 3 pesagens determinar a diferente e se é mais leve ou pesada.

Neste caso $a = 24$ e $b = 3^3 = 27$ pelo que é $a \leq b$ e o problema poderá ter ou não solução; o Teorema 30 só exprime uma condição necessária para haver solução.

Procuremos uma possível solução. Utilizaremos apenas pesagens com igual número de bolas em cada prato. Se, por exemplo, colocassemos 4 bolas no prato da esquerda e 3 no da direita, e a balança a caísse para a esquerda, não tiravamos qualquer conclusão: qualquer uma destas 7 bolas podia ser a mais pesada, a mais leve, ou podiam ser todas iguais!

Primeira pesagem

Temos $a = 24$ e $b = 27$; depois da primeira pesagem ficará $b' = 9$ onde b' representa o valor de b para o problema subsequente. Se colocassemos 1 bola em cada prato e a balança a ficasse equilibrada, restavam 20 hipóteses (10 bolas) para serem decididas em 2 pesagens; como $20 > 9$, isso seria impossível, pelo Teorema 30. A mesma conclusão se tira se usarmos 2 (pois $16 > 9$) ou 3 (pois $12 > 9$) bolas em cada prato. Se usarmos 5 bolas em cada prato e a balança a cair para a esquerda, teríamos que decidir uma de 10 hipóteses (uma das 5 bolas da esquerda pode ser mais a pesada, ou uma das 5 bolas da direita pode ser a mais leve) com 2 pesagens; ora $10 > 9$ e isso é impossível; com 6 bolas em cada prato ainda é pior. Conclusão: teremos que colocar 4 bolas em cada prato, seja $(1, 2, 3, 4) - (5, 6, 7, 8)$. Temos a considerar os casos

1. Fica equilibrado

   Uma das restantes 4 bolas, 9, 10, 11 ou 12 é a diferente; temos 8 hipóteses e $8 < 9$, tudo bem, só pode haver um resultado de pesagem desperdiçado. Façamos seguidamente a pesagem $(9, 10, 11) - (1, 2, 3)$, notando que 1, 2 e 3 são bolas iguais.

   (a) Se cai para a esquerda, ou 9 ou 10 ou 11 é a diferente e é mais pesada ($3 \leq 3$); com uma nova pesagem entre 9 e 10 a situação fica resolvida.

   (b) Se cai para a direita é análogo: ou 9 ou 10 ou 11 é a diferente e é mais leve ($3 \leq 3$); com uma nova pesagem entre 9 e 10 a situação fica resolvida.

(c) Se fica equilibrada, a bola 12 é diferente; comparada com, por exemplo, a bola 1, determinamos se é mais leve ou mais pesada (aqui a balança a não pode ficar equilibrada: é o resultado desperdiçado).

2. Cai para a esquerda

   Há 8 hipóteses: ou 1 ou 2 ou 3 ou 4 é a mais pesada; ou 5 ou 6 ou 7 ou 8 é a mais leve ($8 \leq 9$) Façamos a pesagem $(1, 2, 5) - (3, 4, 6)$.

   (a) Cai para a esquerda: 1 ou 2 é a mais pesada ou 6 é a mais leve; uma nova pesagem decide a situação $(1) - (2)$.

   (b) Cai para a direita; análogo: 3 ou 4 é a mais pesada ou 5 é a mais leve; fazemos a pesagem $(3) - (4)$.

   (c) Fica equilibrada: a mais leve é 7 ou 8; pesemos $(7) - (8)$ (não pode dar igual).

3. Cai para a direita

   Raciocínio análogo ao caso 2.

Exercício 97 *O problema análogo com 13 bolas é insolúvel (por isso é que se diz que 13 dá azar). Provar esse facto.*

10.2 Entropia, informação e minorantes de complexidade

10.2.1 Introdução

Se existem n hipóteses para uma situação e p_i é a probabilidade de o facto i ser verdadeiro a entropia deste estado de conhecimento é o seguinte valor que nunca é negativo

$$S = -\sum_{i=1}^{n} p_i \log p_i$$

Esta entropia mede a falta de informação ou incerteza deste estado de conhecimento; admitimos que os logaritmos são na base 2 embora isso não seja importante aqui. Uma entropia grande significa que é necessária uma grande quantidade de informação para descrever a situação. Uma entropia nula quer basicamente dizer que o conhecimento da situação é completo.

Vejamos em 2 situações o valor tomado pela entropia.

– Se temos a certeza que a hipótese 1 é válida, $p_1 = 1$ e $p_i = 0$ para $i \geq 2$. Resulta que o valor limite de S quando todos os p_i, com $i \geq 2$, tendem para 0 (e, consequentemente, p_1 tende para 1)

$$S = 1 \times \log 1 = 0$$

levando em consideração que $\lim_{p \to 0} p \log p = 0$.

– Se qualquer das n hipóteses é igualmente provável, $p_i = 1/n$ para todos os $1 \leq i \leq n$. Temos pois

$$S = -n \frac{1}{n} \log \frac{1}{n} = -\log \frac{1}{n} = \log n$$

Por exemplo, se $n = 16$, vem $S = 4$; a entropia (usando o logaritmo na base 2) mede, o número de bits de informação necessários para especificar completamente uma das hipóteses.

A fórmula da entropia não "caiu do céu". Shannon em *the Mathematical Theory of Information* demonstra que a *única* função $S(p_1, \cdots, p_n)$ que satisfaz os seguintes pressupostos razoáveis

– S é contínua em todos os p_i.

– Se todos os $p_i = 1/n$, a função $S(n)$ deverá crescer monotonicamente com n. Mais hipóteses (=acontecimentos) representam mais incerteza.

– Se uma escolha se desdobrar em duas escolhas sucessivas, a função ao original S deve ser a soma pesada dos S de cada escolha (ver o trabalho referido de Shannon para mais pormenores).

é do tipo

$$S = -k \sum_{i=1}^{n} p_i \log p_i$$

onde $k > 0$.

10.2.2 Informação e os problemas de pesagens

Uma pesagem do tipo considerado pode ter 3 resultados; seja:

n o número de hipóteses possíveis antes da pesagem.

n_1, n_2 e n_3 os números de hipóteses que restam após cada pesagem.

Supondo probabilidades iguais para as situações possíveis, a entropia é diminuida de, respectivamente, $\log(n/n_1)$, $\log(n/n_2)$ e $\log(n/n_3)$. Ora, como pretendemos limitar o número máximo de pesagens em todas as situações, a situação mais favorável seria quando a divisão fosse disjunta

(em hipóteses) e igual, isto é $n_1 = n_2 = n_3 = n/3$. Nesta hipótese a pesagem daria um "trit" de informação:

$$\text{Um trit} = \log_2 3 \text{ bits} \approx 1.585 \text{ bits}$$

Em termos de informação expressa em trits, a possível solução do problema das 12 bolas (ver o teorema30) resulta da desigualdade:

$$3 \text{ trits} \geq \log_3 24 \text{trits} \approx 2.893 \text{ trits}$$

Nota: Nota: a desigualdade $a \leq b$ deve ser válida em todos os nós da árvore.

Exercício 98 *Construir a árvore da solução dada para as 12 bolas, representando em cada nó:*

- *A pesagem efectuada.*
- *A entropia (antes da pesagem) expressa em trits.*
- *A entropia máxima, dado o número que falta de pesagens (0, 1, 2 ou 3).*

10.3 Minorantes de algoritmos de ordenação

Anteriormente determinamos minorantes do número de pesagens que é necessário efectuar com vista a determinar qual a bola de um determinado conjunto que é diferente das outras. Vamos agora aplicar o mesmo tipo de técnicas – baseadas na Teoria da Informação – para determinar minorantes de complexidade de problemas de ordenação. O leitor deve rever o conceito de "modelo externo dos dados", ver página 37.

Vejamos agora um problema que tem a ver com os importantes algoritmos de ordenação.

Problema (Ordenação)
Sejam n bolas de pesos diferentes; pretende-se colocá-las por ordem crescente de pesos usando apenas operações de comparação entre 2 bolas, isto é, pesagens com uma bola num prato e outra noutro. Note-se que a balança a nunca fica equilibrada uma vez que não existem 2 bolas de igual peso.

Aplicando directamente o Teorema 30 temos.

O número de hipóteses possíveis é $a = n!$.

O número de situações que se podem discriminar com c comparações é $b = 2^c$.

Resulta que qualquer algoritmo de ordenação que obtenha a informação através de comparações deve satisfazer $a \leq b$, ou seja,

$$2^c \geq n!$$

ou ainda

$$c \geq \lceil \log(n!) \rceil$$

Exercício 99 *Determine minorantes de c para os primeiros valores de n (por exemplo entre 1 e 6) e procure algoritmos de ordenação que se aproximem tanto quanto possível desses valores (no caso mais desfavorável).*

Usando a fórmula de Stirling como aproximação para o factorial

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

temos (podiamos ser mais rigorosos; o sinal $\approx$ pode ser substituido por $>$ e este facto pode ser utilizado no que se segue).

$$2^c \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

ou seja (os logaritmos são, como é usual, na base 2)

$$c \geq \frac{1}{2} \log(2\pi n) + n \log n - n \log e$$

Vemos pois que qualquer algoritmo de ordenação baseado em comparações deve fazer (no pior caso) pelo menos cerca de $n \log n$ comparações, isto é deve ser pelo menos de ordem $O(n \log n)$.

Exercício 100 *Considere o algoritmo de ordenação "mergesort". Compare para* $n = 1, 2, 4, \ldots, 1024$ *o minorante teórico do número de comparações (*$\lceil \log(n!) \rceil$*) com o majorante número de comparações calculado para o algoritmo (ver apontamentos teóricos de Análise de Algoritmos).*

Concluimos pois que, em termos de ordens de grandeza (a afirmação não é válida se considerarmos os valores exactos do número de comparações), são conhecidos algoritmos de ordenação óptimos. Um exemplo é o "heapsort"; o "quicksort" não é óptimo, uma vez que as nossas conclusões são em termos do comportamento no pior caso (para o qual o "quicksort" é $O(n^2)$).

O teorema 30 dá para cada problema um limite mínimo da complexidade dos algoritmos respectivos; pode ser expresso, por outras palavras, da forma seguinte: o algoritmo tem que obter pelo menos tanta informação quanta é necessária para distinguir o caso particular do problema em questão. Para certos problemas — como o da ordenação — existem algoritmos próximos deste limite teórico; para outros, os melhores (teoricamente possiveis) algoritmos estão muito longe dele e há que recorrer a outros métodos, em geral mais complicados, para obter melhores minorantes.

10.4 Algoritmos de ordenação em que o custo é o número de trocas

Vamos considerar o problema de ordenar um vector de n elementos, usando como medida de custo o *número de trocas* que são efectuadas. O vector que se pretende ordenar só pode ser alterado através de trocas. Cada troca tem o custo 1; as outras operações têm o custo 0.

Um exemplo prático seria o seguinte: temos n livros numa prateleira e pretendemos colocá-los por ordem. Só há um custo: a troca entre 2 livros; todos os outras operações (como as comparações) têm custo 0.

Quantas trocas são necessárias?

Teorema 31 *$n-1$ trocas são suficientes.*

Dem. Basta exibir um algoritmo que efectue $n-1$ trocas. O leitor pode facilmente verificar que o método de ordenação "selecção do mínimo" está nessa situação. □

Teorema 32 *$n-1$ trocas são necessárias.*

Antes de vermos a demonstração deste resultado vamos definir o que é o grafo associado ao vector que se pretende ordenar.

Definição 19 *O grafo dirigido associado a um vector com n elementos (distintos) tem n vértices e tem um arco (i,j) sse, com a ordenação, o elemento que está na posição i do vector vai para a posição j.*

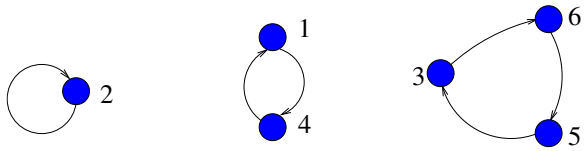
Por exemplo, o grafo correspondente ao vector `[40,20,600,10,30,50]` pode ser representado por

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 6 & 1 & 3 & 5 \end{pmatrix}$$

Nota importante. Os inteiros nos diagramas e nos grafos designam sempre posições (e nunca valores); por exemplo, o elemento que está na posição 5 vai para a posição 3.

A tabela anterior também pode ser representado pelo "produto de ciclos"

$$(1,4)(2)(3,6,5)$$



Repare-se que um vector está ordenado se o grafo associado tiver n ciclos, cada um deles com um só elemento.

O que acontece ao grafo se trocarmos um par de valores do vector? Sejam i e j as posições em que se encontram inicialmente esses valores.

- Se essas posições, i e j, estão no mesmo ciclo, esse ciclo desdobra-se em 2 ciclos. Exercício. demonstre esta afirmação, começando por trocar no exemplo dado as posições 3 e 5, bem como as posições 1 e 4.
- Se essas posições, i e j, estão em ciclos diferentes, esses 2 ciclos juntam-se num só ciclo. Exercício. demonstre esta afirmação.

Vamos agora provar o Teorema anterior.

Dem. Considere-se um vector cujo grafo tenha um só ciclo, por exemplo

`[2,3,4,...,n,1]`

Exercício. Verifique que só há 1 ciclo. Temos então

- Cada troca aumenta no máximo o número de ciclos de 1 (também pode reduzir de 1).
- O vector inicial dado tem 1 ciclo.
- O vector ordenado tem n ciclos.

Então qualquer sucessão de trocas que ordene o vector `[2,3,4,...,n,1]` tem, pelo menos, $n-1$ trocas. □

10.5 Minorantes de algoritmos de determinação do maior elemento

Um outro problema, mais simples que o da ordenação é o seguinte.

Problema (Máximo)

Determinar o maior de entre n valores distintos.

Neste caso a aplicação do Teorema 30 fornece um minorante de complexidade muito fraco: são precisas pelo menos $\log n$ comparações (o número de soluções possíveis é n); ora é possível mostrar que são necessárias $n-1$ comparações!

Exercício 101 *Temos n bolas; uma pesa 1, outra 2,... e a outra 2^{n-1} Mostre que é possível determinar a mais pesada em não mais de cerca de $\log n$ pesagens (aproximando-se do limite dado pelo teorema).*

Exercício 102 *Analise a complexidade mínima do seguinte problema. Existem n elementos distintos v_1, v_2,...,v_n ordenados por ordem crescente; é dado um valor x e pergunta-se: qual o menor i tal que $v_i > x$ (sendo $i = 0$ se x é menor que todos os elementos)? Por outras palavras, onde deveria x ser colocado na lista dos elementos?*

Notas sobre a solução

Aqui existem $n+1$ respostas possíveis; o teorema diz-nos que o número de comparações deve satisfazer: $2^c \geq n+1$, ou seja $c \geq \lceil \log(n+1) \rceil$.

Existe um algoritmo — o da pesquisa binária — que faz exactamente este número de comparações; não pode haver melhor!

Exercício 103 *Na seguinte situação determine (i) quantas comparações são necessárias pelo Teorema 30? (ii) E na realidade? (iii) Qual a entropia da situação inicial?*

- *A "pensa" num inteiro secreto entre 1 e 1000; B tenta adivinhar qual é, dizendo de cada vez um número; A responde: é maior, menor ou é esse.*

Teorema 33 (Majorante da complexidade) *$n-1$ comparações são suficientes para determinar o maior de n elementos distintos.*

Dem. Existe um algoritmo muito simples de determinação do maior elemento que estabelece este majorante. □

É interessante verificar que $n-1$ comparações são também necessárias!

Teorema 34 (Minorante da complexidade) *São*[1] *necessárias $n-1$ comparações para determinar o maior de n elementos distintos.*

Dem. Consideremos um qualquer algoritmo que determina o maior dos n elementos. Seja G um grafo não dirigido constituído por n vértices (associados aos elementos do vector) e com um arco entre os vértices i e j sse os elementos i e j do vector foram comparados entre si. Esse grafo evolui ao longo da execução do algoritmo.

No início, o grafo tem n componentes conexos, cada um com 1 vértice. Cada vez que se faz uma comparação o número de componentes conexos

- ou se mantém, se i e j estão no mesmo componente conexo,
- ... ou diminui de uma unidade, se i e j estão em componentes conexos distintos.

Suponhamos que havia 2 ou mais componentes no final da execução do algoritmo e seja x a resposta do algoritmo; seja ainda C_1 o componente a que pertence x e C_2 outro componente. O valor x não foi comparado com qualquer elemento de C_2 (não houve qualquer comparação entre os elementos de C_1 e os elementos de C_2). Assim, se aumentarmos todos os valores de C_2 de um mesmo valor suficientemente grande, a resposta x está errada, uma vez que o maior elemento está em C_2; e o resultado das comparações efectuadas pelo algoritmo é exactamente o mesmo.

Assim, no final o grafo tem que ter exactamente 1 componente conexa, pelo que tem que efectuar pelo menos $n-1$ comparações. □

10.6 Determinação do segundo maior elemento

Consideremos agora o problema de determinar o segundo maior elemento de um vector com n elementos distintos.

Um minorante de complexidade é fácil de estabelecer.

Teorema 35 (Minorante da complexidade) *São necessárias $n-1$ comparações para determinar o segundo maior elemento de entre n elementos distintos.*

[1] Usando o modelo de custos do pior caso, como sempre neste Capítulo. Isto é, poderíamos reformular o enunciado: "Para qualquer n podem ser necessárias...".

Dem. A demonstração do Teorema 34 é aplicável. □

Relativamente ao majorante de complexidade temos o seguinte resultado.

Teorema 36 (Majorante fraco da complexidade) *Bastam* $2n-3$ *comparações para determinar o segundo maior de entre* n *elementos distintos.*

Dem. Podemos determinar o segundo maior elemento de entre n elementos distintos pelo seguinte processo: (i) determinamos o maior elemento ($n-1$ comparações); (ii) percorremos de novo o vector com exclusão do maior elemento para obter o segundo maior elemento ($n-2$ comparações). □

Temos um intervalo de desconhecimento: entre $n-1$ e $2n-3$. O seguinte resultado reduz substancialmente esse intervalo, apresentando um algoritmo que efectua pouco mais que $n-1$ comparações.

Teorema 37 (Majorante forte da complexidade) *Bastam* $n + \log n - 2$ *comparações para determinar o segundo maior de entre* n *elementos distintos.*

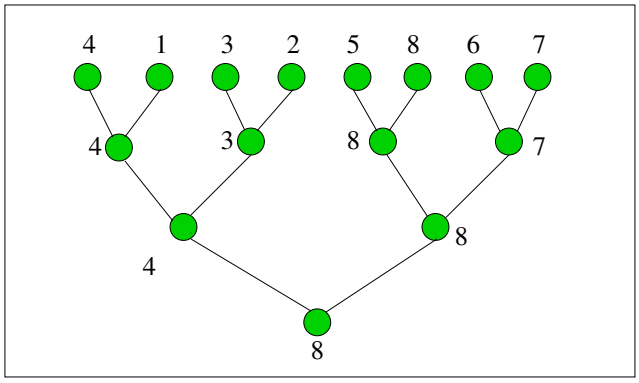
Dem. Consideremos o seguinte algoritmo em que supomos que n é uma potência de 2.

1) Os elementos são comparados aos pares ($n/2$ comparações);

2) Os maiores elementos de cada par são comparados aos pares ($n/4$ comparações);

... ... (até encontrar o maior elemento)

O número total de comparações é $n/2+n/4+\ldots+1 = n-1$ (demonstre esta igualdade). A figura seguinte ilustra a árvore resultante para o vector `[4,1,3,2,5,8,6,7]`.



Como o maior elemento foi necessariamente comparado com o segundo maior elemento (esta observação é atribuída a Lewis Carroll!), basta percorrermos a árvore da raíz para o topo com vista a determinar o segundo maior elemento; no exemplo da figura esse elemento é $\max\{4,7,5\} = 7$. O número de comparações é $\log n - 1$ (demonstre).

O número total de comparações é $(n-1)+(\log n-1) = n+\log n-2$. Quando n não é uma potência de 2, este resultado não é válido, mas o número de comparações é limitado por $n+\log n+c$ onde c é uma constante. □

10.7 Minorantes do problema de "Merge"

Consideremos agora o algoritmo do "merge".

Problema ("Merge")

São dadas 2 sequências de ordenadas de inteiros, uma com n e a outra com m elementos.

Pretende-se obter a sequência ordenada constituída pelos $m+n$ elementos.

Um algoritmo de resolver este problema é baseado na aplicação iterativa do seguinte passo. Os dois menores elementos das sequências dadas são comparados; o menor deles é retirado e colocado na sequência resultado. Quando uma das sequências dadas "chegar ao fim", o resto da outra é "passado" para a sequência resultado. Supondo todos os elementos distintos, é fácil ver que são feitas no máximo $m+n-1$ comparações.

Vejamos se este algoritmo se aproxima do limite teórico; o número de situações possíveis diferentes entre as duas sequências é:

$$\binom{m+n}{n}$$

Este número, igual a $(m+n)!/(n!m!)$, pode ser aproximado usando a fórmula de Stirling; vamos supor apenas o caso $m=n$.

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \approx (\pi n)^{-1/2} 2^{2n}$$

Sendo c o número de comparações efectuadas, deverá ser pois:

$$2^c \geq \left\lceil (\pi n)^{-1/2} 2^{2n} \right\rceil$$

ou seja:

$$c \geq \left\lceil 2n - \frac{1}{2}\log(\pi n) \right\rceil$$

Ora, como dissemos, o algoritmo do "merge" faz neste caso $2n-1$ comparações na pior hipótese. Vemos que, para n grande, o algoritmo é quase óptimo. Por exemplo, para $n=100$, o algoritmo faz 199 comparações, enquanto o minorante (confiando na aritmética do yap) é $\lceil 195.852 \rceil = 196$.

10.8 Conectividade de grafos

Consideremos o problema de determinar se um grafo não dirigido $G = (V, E)$ com n vértices é conexo.

Usamos o modelo exterior dos dados; o algoritmo faz perguntas do tipo "$(i, j) \in E$?", isto é, "existe o arco $i - j$?".

O seguinte resultado é trivial.

Teorema 38 *Um majorante de complexidade é* $n(n-1)/2$.

Dem. Na verdade, um grafo não dirigido fica inteiramente caracterizado pelos $n(n-1)/2$ elementos (0's ou 1's) abaixo da diagonal principal da matriz das incidências[2]. Assim, um algoritmo que efectue $n(n-1)/2$ perguntas pode obter informação completa sobre o grafo e determinar se ele é conexo. □

Vamos agora ver que é realmente necessário ler todos esses $n(n-1)/2$ bits para responder sempre de forma correcta.

Teorema 39 *Um minorante de complexidade é* $n(n-1)/2$.

Dem. Consideremos um qualquer algoritmo de conectividade que obtém informação sobre o grafo da forma indicada. Vamos mostrar que se o algoritmo não fizer todas as $n(n-1)/2$ perguntas "$(i, j) \in E$?", há sempre um grafo a que responde erradamente, isto é, que diz "conexo" quando o grafo não é conexo, ou vice-versa. Esse grafo onde o algoritmo falha é construído (por um "adversário"...) da forma seguinte

> Regra. A resposta a "$(i, j) \in E$?" é `não` a não ser que o grafo já desvendado[3] ficasse definitivamente desconexo[4].

Há uma propriedade (um invariante) que vai servir de base a esta demonstração.

> Invariante. Para qualquer par (i, j) ainda não desvendado, o grafo já desvendado não tem caminho entre i e j.
>
> Prova. Suponhamos que havia caminho no grafo já desvendado entre i e j e seja (i', j') o último arco já desvendado (resposta `sim`) desse caminho. Mas então, na altura em que se respondeu a "$(i', j') \in E$?" com `sim` não se seguiu a regra indicada atrás, uma

[2] Para efeitos de conectividade a existência de um arco entre um vértice e ele próprio é irrelevante.

[3] Entende-se por "grafo já desvendado" o grafo de n vértices que apenas ter arcos $i \to j$ se a resposta a uma pergunta "$(i, j) \in E$?" foi `sim`.

[4] Isso pode ser determinado analisando a conectividade do grafo que resulta de completar o grafo já desvendado com ligações entre todos pares ainda não desvendados.

vez que deveríamos ter respondido com `não`, pois haveria a possível ligação (i, j) que manteria (mais tarde) a conectividade do grafo já desvendado. □

Se um algoritmo não faz todas as $n(n-1)/2$ perguntas, consideremos um (i, j) não desvendado.

- Se o algoritmo diz "`conexo`", nós (o adversário...) apresentamos, como contra-exemplo, o grafo já desvendado sem qualquer outro arco. Na verdade, pelo invariante demonstrado atrás, não existe caminho entre i e j e o grafo não é conexo.
- Se o algoritmo diz "`não conexo`", essa resposta está errada, pois contradiz a estratégia de construção do grafo por parte do adversário.

Assim, um algoritmo que não faça todas as $n(n-1)/2$ perguntas não está correcto. □

Capítulo 11

Apêndices

11.1 Apêndice: Alguns resultados úteis

Apresentam-se alguns resultados matemáticos úteis. O aluno deverá ser capaz de demonstrar a maioria deles.

Somas aritméticas

Se a diferença entre cada elemento e o anterior é uma constante, tem-se $a_1 + a_2 + \cdots + a_n = n(a_1 + a_n)/2$. Em particular $1 + 2 + 3 + \cdots + n = n(n + 1)/2$.

Somas geométricas

Se o quociente entre cada elemento e o anterior é uma constante c, tem-se $a_1+a_2+\cdots+a_n = a_1\frac{c^n-1}{c-1}$. Em particular $1 + 2 + 4 \cdots + 2^n = 2^{n+1} - 1$.

Soma dos quadrados

$1^2+2^2+3^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$. Mais geralmente, a soma das potências p, $1^p+2^p+3^p+\cdots+n^p$ é um polinómio de grau $p + 1$ cujos coeficientes se determinam sem dificuldade para cada valor de p.

Soma harmónica

$1 + 1/2 + 1/3 + 1/4 + \cdots + 1/n = \ln n + \gamma + O(1/n)$ onde $\gamma = 0.57\ldots$ é a constante de Euler ([5]).

Aproximação de Stirling do factorial

$n! = \sqrt{2\pi n}(n/e)^n(1 + O(1/n))$, ver [5].

Logaritmos

As seguintes desigualdades resultam directamente da noção de logaritmo. $a^{\log_a x} = x$, $a^{\log_b x} = x^{\log_b a}$, $\log_a b = 1/\log_b a$, $\log_a x = \log_b x/\log_b a$.

11.2 Apêndice: Implementações do "quick sort" em Haskell, Prolog e Python

Apresentamos implementações possíveis do quick sort nas linguagens Haskell, Prolog e Python.

Continuamos a supor que todos os elementos do vector são distintos. Dado estas implementações envolverem a concatenação de listas, deve ser estudado em que medida é esse facto tem influência na complexidade dos algoritmos.

```
Haskell
-------

  qs []     = []
  qs (s:xs) = qs [x|x <- xs,x < s] ++ [s] ++ qs [x|x <- xs,x > s]



Prolog
------

  qs([], []).
  qs([H | L], S) :- partition(H, L, Left, Right),
                    qs(Left, SL),
                    qs(Right, SR),
                    append(SL, [H | SR], S).

  partition(Piv,[],[],[]).
  partition(Piv,[H|L],[H|Left],Right) :- H < Piv,
                                         partition(Piv, L, Left, Right).
  partition(Piv,[H|L], Left, [H|Right]) :- H > Piv,
                                           partition(Piv, L, Left, Right).



Python
------

  def qs(a):
      if len(a)<=1:
          return a
      return qs([x for x in a if x<a[0]]) + [a[0]] +
             qs([x for x in a if x>a[0]])
```

11.3 Apêndice: Algoritmo eficiente de potenciação modular

```
//-- calcula m^x (mod n)
int mpow(int m,int x,int n){
  int t,t2;
  if(x==0)
    return 1;
  t=mpow(m,x/2,n);
  t2=(t*t)%n;             (*)
  if(x%2==0)
    return t2;
  else
    return (t2*m)%n;      (*)
}
```

Exercício 104 *Determine o número de multiplicações que são efectuadas pelo algoritmo (linhas* `(*)`*) em função da representação binária de x (mais em pormenor, em função do número de 0's e do número de 1's da representação binária de x).*

11.4 Apêndice: Algoritmo de Rabin-Miller (teste de primalidade)

```
# Linguagem: python
# Obtido de: http://snippets.dzone.com/posts/show/4200

import sys
import random

def toBinary(n):
  r=[]
  while (n>0):
    r.append(n%2)
    n=n / 2
  return r

def test(an):
  """
  test(an) -> bool Tests whether n is complex.
  Returns:
    - True if n is complex.
    - False, if n is probably prime.
  """
  b=toBinary(n-1)
  d=1
  for i in xrange(len(b)-1,-1,-1):
    x=d
    d=(d*d)%n
    if d ==1 and x !=1 and x != n-1:
      return True # Complex
    if b[i]==1:
      d=(d*a)%n
  if d !=1:
    return True # Complex
  return False # Prime

def MillerRabin(ns=50):
  """
    MillerRabin(ns=1000) -> bool Checks whether n is prime or not

    Returns:
      - True, if n is probably prime.
      - False, if n is complex.
  """
  for j in xrange(1,s+1):
    a=random.randint(1,n-1)
    if (test(a, n)):
      return False # n is complex
  return True # n is prime

def main(argv):
  print MillerRabin(int(argv[0]),int(argv[1]))

if __name__ == "__main__":
  main(sys.argv[1:])
```

11.5 Apêndice: Algoritmo do "radix sort" em python

```
# comprimento do alfabeto
nalfa=256

# radix sort; (menos significativo primeiro)
def radixsort(lista):
    m=max(map(len,lista))
    balde=[0]*nalfa
    for i in range(0,nalfa):
        balde[i] = Fila()
    for i in range(m-1,-1,-1):
        rsort(lista,i,m,balde)
    return lista

#-- ordena segundo simbolo d
def rsort(lista,d,m,balde):
    for x in lista:
        # x pode ter comprimento < d!
        if d<len(x):
            indice=ord(x[d])
        else:
            indice=0
        balde[indice].entra(x)
    ilista=0
    for d in range(nalfa):
        while not balde[d].vazia():
            lista[ilista]=balde[d].sai()
            ilista += 1
    return lista

class Fila:
    def __init__(self):
        self.fila=[]
    def vazia(self):
        return self.fila==[]
    def entra(self,x):
        self.fila.append(x)
    def sai(self):
        return self.fila.pop(0)

#-----Exemplo de uso
>>> a=["bbc","abc","acb","ccb","z","za","ccb"]
>>> print radixsort(a)
    ['abc', 'acb', 'bbc', 'ccb', 'ccb', 'z', 'za']

>>> b=["b","a","c"]
>>> print radixsort(b)
    ['a', 'b', 'c']
```

Bibliografia

[1] Gilles Brassard, Paul Bratley, *Fundamentals of Algorithmics*, Prentice Hall 1996. Em especial: Capítulo I: (generalidades), Capítulo II, páginas: 37–43, 65–75.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms* (second edition) MIT Press and McGraw-Hill, 2001s,
url=`http://serghei.net/docs/programming/Cormen/toc.htm`.

[3] Udi Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[4] R. Sedgewick, P. Flajolet, *An Introduction to the Analysis of Algorithms*. Pearson, 1998.

[5] Donald Knuth, *The Art of Computer Programming*, Volume 1 (Fundamental Algorithms), third edition, Addison-Wesley, 1997.

[6] Donald Knuth, *The Art of Computer Programming*, Volume 3 (Sorting and Searching), second edition, Addison-Wesley, 1997.

[7] Michael Luby, *Pseudorandomness and Cryptographic Applications*, Princeton Computer Science Notes, 1996.

Nota. Na bibliografia em cima as referências [2, 1, 3] são, por esta ordem de importância, livros com interesse para a disciplina; estes livros existem na biblioteca do departamento. Durante o semestre foram disponibilizadas outras referências que não mencionamos aqui. As restantes referências não têm interesse directo para a disciplina, sendo apenas "ocasionais".