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.*
- `m` $=$ `n`*; como corolário,* `w[]` *"não tem elementos que não estão em* `v[]`*".*
- $\forall i,\ 1 \leq i \leq n-1$: `w`$[i] \leq$ `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"

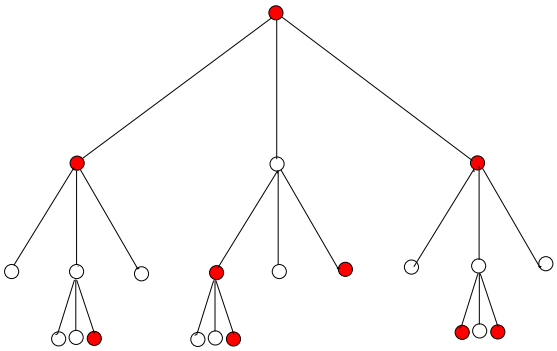
Podemos dizer que os algoritmos de ordenação descritos em (5.1.1), (5.1.3), (5.1.5), (5.2.1) e (5.2.2) são baseados na indexação pelos valores a ordenar (ou parte desses valores) e não por comparação entre esses valores. Existe uma estrutura de informação, a `trie` que é também baseada nos valores que 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	$b-a-k$
$a, a+1 \ldots\ldots\ldots\ldots, a+k-1$	$a+k$	$a+k+1, a+k+2, \ldots\ldots\ldots\ldots, b$

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)

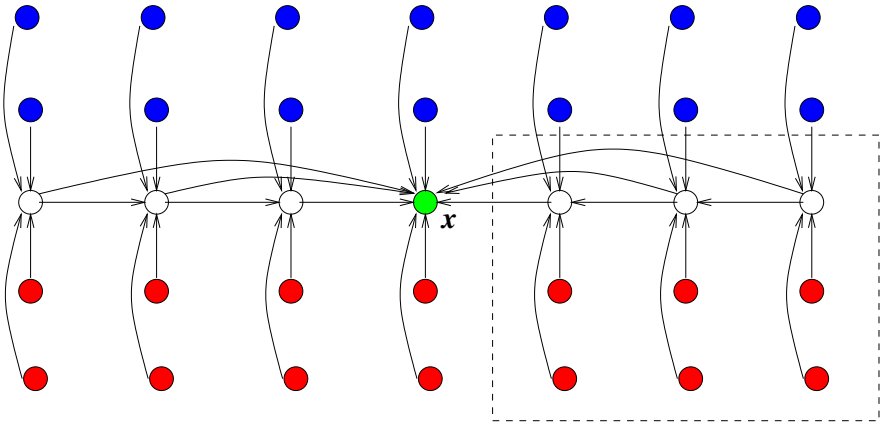
Para se conseguir um tempo de ordem $O(n)$ no pior caso é essencial conseguir obter um elemento x do vector que tem, qualquer que seja o vector com n elementos, pelo menos fn elementos menores que ele e pelo menos fn elementos maiores que ele, onde f é um número real positivo conveniente. Apresentamos 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: $t(7n/10)$

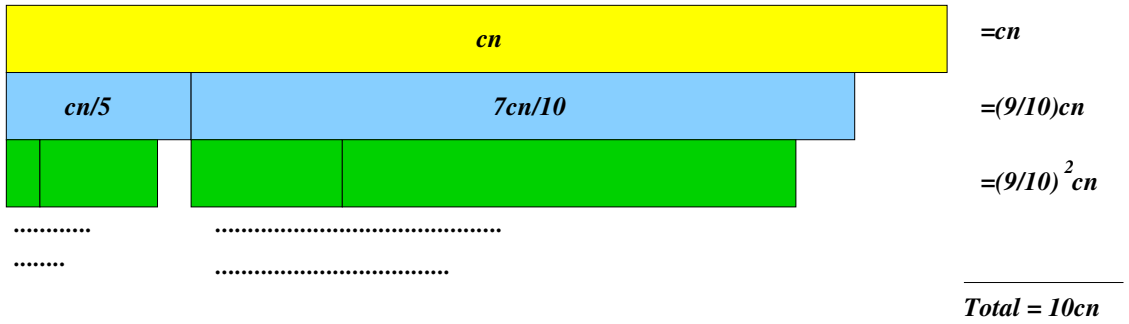
O termo geral da recorrência é

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

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

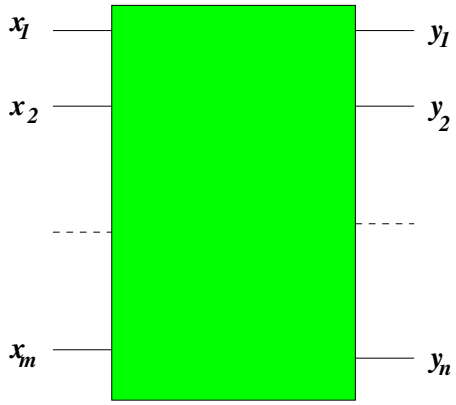
Os circuitos são uma alternativa não uniforme aos algoritmos. Enquanto um algoritmo pode ter dados de qualquer comprimento, um circuito tem uma aridade (número de argumentos) fixa. Assim, por exemplo, uma alternativa a um algoritmo de ordenação é uma família infinita C_1, C_2,... de circuitos em que 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

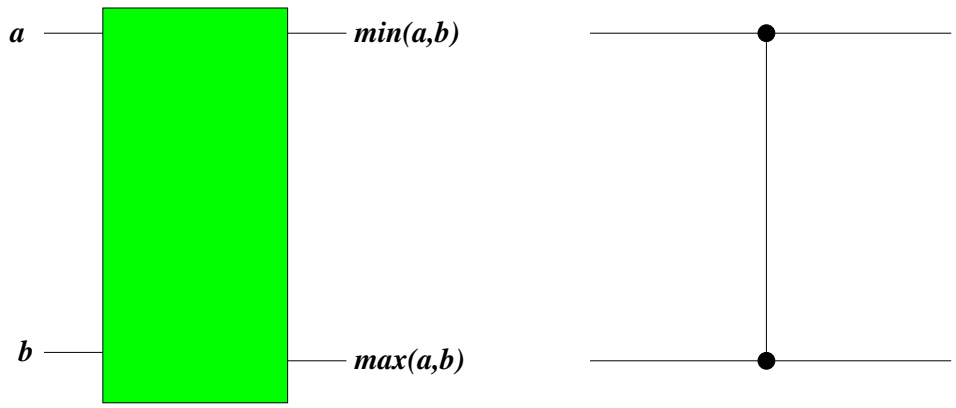
Plano desta secção
O objectivo principal desta secção é implementar uma[2] rede de ordenação eficiente. Essa rede será baseada no "merge sort". Para tal fim seguiremos os seguintes passos

- (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.

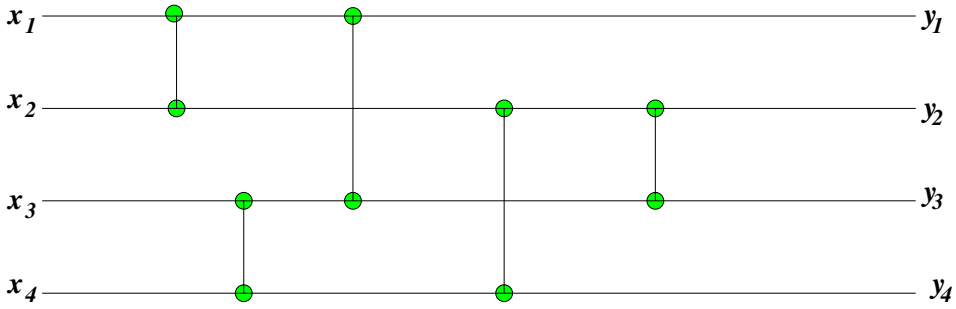


As saídas são: o menor e o maior dos valores de entrada; não sendo indicado expressamente o contrário, o menor dos valores fica em cima e o maior em baixo (como na figura anterior). Estes componentes são representados por "|" (como na figura anterior à direita) ou "↓". Quando pretendermos que o maior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

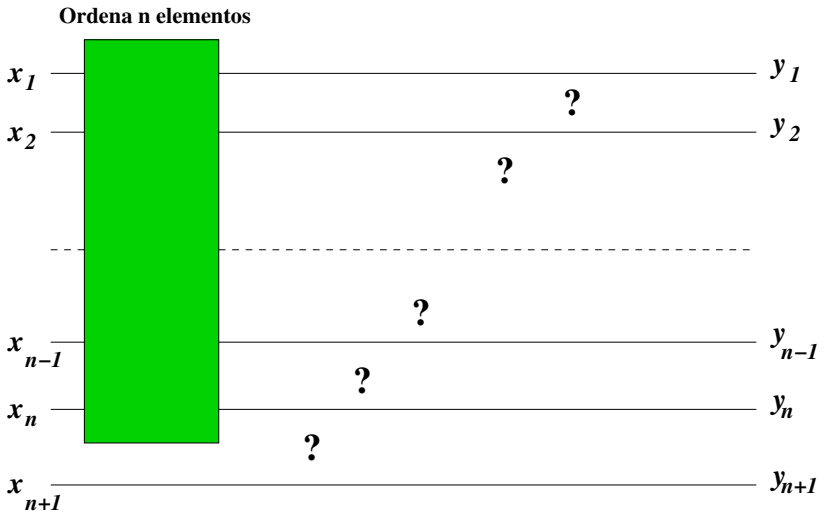


Exercício 59 *Considere as entradas $x_1 = 8$, $x_2 = 5$, $x_3 = 1$, $x_4 = 6$.*

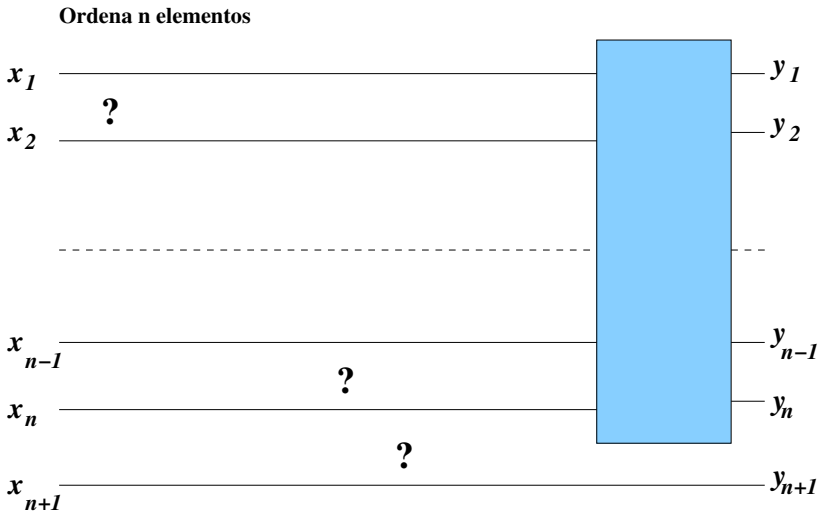
1. *Quais os valores de y_1, y_2, y_3 e y_4?*
2. *Qual é a profundidade do circuito?*
3. *Quantas comparações são efectuadas?*
4. *Indique os valores que ficam definidos ao fim de t unidades de tempo com $t = 0, 1, 2, \ldots$.*

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 $n+1$?*



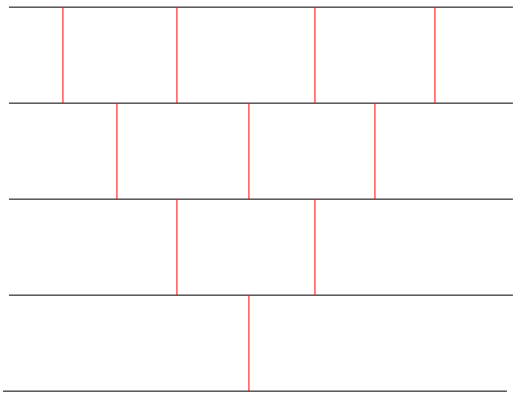
Exercício 62 *Implemente uma rede de ordenação baseada no método de ordenação da bolha ("bubble sort"). Use um método semelhante ao do exercício anterior, começando contudo por colocar o maior elemento na linha $n+1$.*



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)

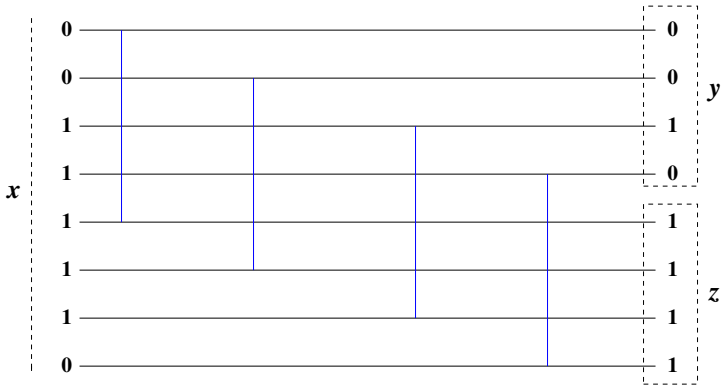
que é, em termos gerais, o seguinte: para valores de um determinado parâmetro inferiores a um certo valor a, a probabilidade assimptótica de uma determinada propriedade ocorrer é 0, sendo essa probabilidade 1 se o parâmetro for maior que a (ou o inverso).

[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.

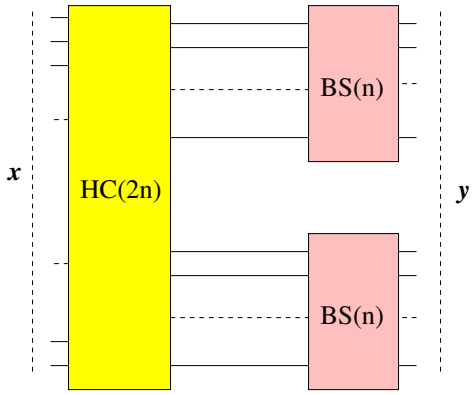
Vamos caracterizar o ordenador bitónico (BS, "bitonic sorter") de forma indutiva, mostrando ao mesmo tempo, por indução, que o circuito construído ordena qualquer sequência de 0's e 1's. O BS(n) vai ser caracterizado para valores de n que são potências de 2.

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.

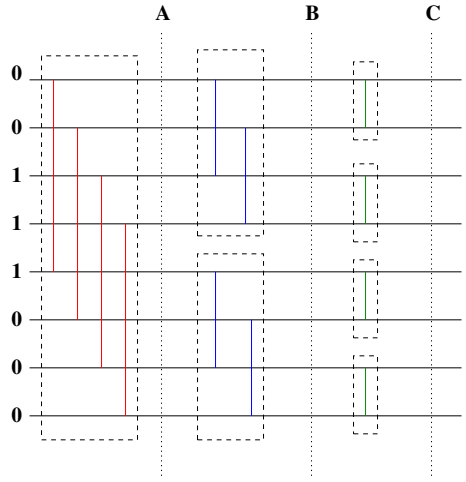


Dado que

- 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. □

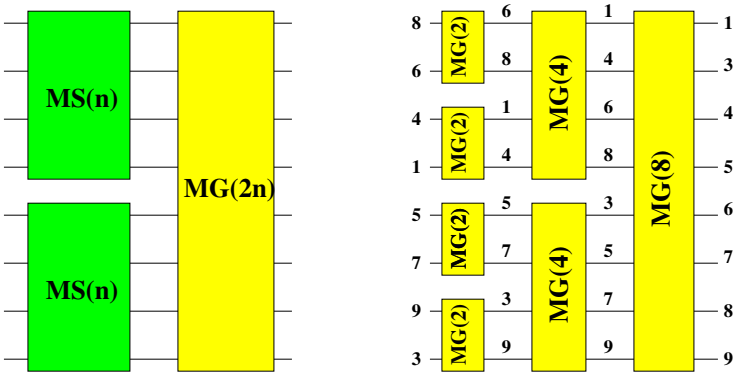
Exercício 68 *A figura seguinte descreve completamente o BS(8). Explique como foi obtida à custa da regra recursiva de construção de BS(n), ver a prova anterior. Indique os valores nas linhas verticais A, B e C.*



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"

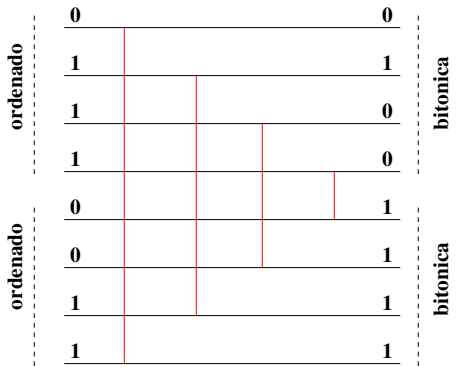
Vamos construir uma rede que recebe uma entrada de $2n$ valores (0 ou 1) constituída por 2 partes, os primeiros n e os últimos n valores, cada uma das quais ordenadas e produz à saída o resultado do "merge" dessas 2 partes. Vamos trabalhar apenas com 0's e 1's; o Princípio 0/1 garante-nos que a rede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

Reparemos que

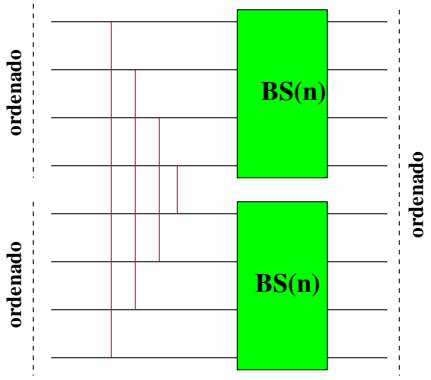
- Uma sequência ordenada é necessariamente da forma 0^i1^j, portanto bitónica
- O "merge" das sequências $w = 0^i1^j$ e $z = 0^k1^l$ é $0^{i+k}1^{j+l}$
- Se w e z estão ordenados, a aplicação da rede HC a wz^t (onde z^t representa z escrito "ao contrário") produz 2 sequências bitónicas em que todo o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