

T4. Tópicos sobre Algoritmos de Ordenação

[Copiar link](#)

Algoritmo de ordenação Quicksort

Trata-se de um algoritmo de *divisão e conquista*, tal como o merge sort.

1. **Divisão:** *partição* do vector $A[p..r]$ em dois sub-vectores

- $A[p..q-1]$ e
- $A[q+1..r]$

tais que todos os elementos do primeiro (resp. segundo) são $\leq A[q]$ (resp. $\geq A[q]$).

Uma função auxiliar de partição recebe a sequência $A[p..r]$, executa a sua partição “in place” usando o último elemento do vector como *pivot*, e devolve o índice q . Note-se que um dos sub-vectores pode ser vazio.

2. **Conquista:** ordenação recursiva dos dois vectores

3. **Combinação:** nada a fazer!

Enquanto no *merge sort* o trabalho era feito na fase de combinação (fusão ordenada), aqui é feito na fase de divisão (partição), que claramente executa em tempo $\Theta(N)$. Vejamos uma implementação possível:

```
1 int partition (int A[], int p, int r)
2 {
3     x = A[r];
4     i = p-1;
5     for (j=p ; j<r ; j++)
6         if (A[j] <= x) {
7             i++;
8             swap(A, i, j);
```

```
9     }
10    swap(A, i+1, r);
11    return i+1;
12 }
```

EXERCÍCIO: Identifique um invariante apropriado para o ciclo desta função de partição.

Tal como no merge sort, depois de definida a função auxiliar, o algoritmo de ordenação é facilmente implementado. A seguinte função ordena o array entre os índices p e r .

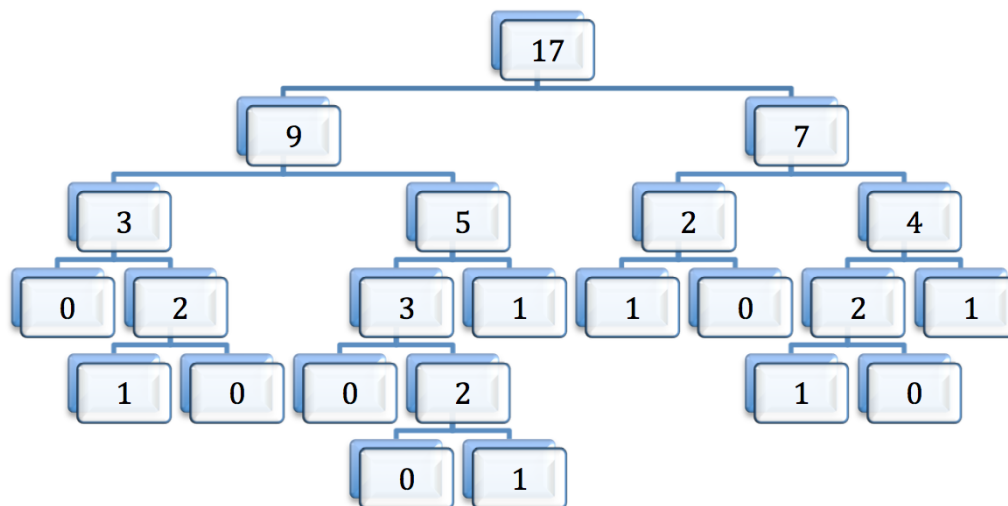
```
1 void quicksort(int A[], int p, int r)
2 {
3     if (p < r) {
4         q = partition(A, p, r);
5         quicksort(A, p, q-1);
6         quicksort(A, q+1, r);
7     }
8 }
```

Vejamos uma simulação de execução do algoritmo. Mostra-se:

- a azul os elementos usados como pivots
- a verde, elementos que já foram pivot e foram colocados na posição final
- a vermelho, elementos que são casos de paragem (função chamada com um só elemento)

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Ao contrário do algoritmo merge sort, a árvore de recursividade do quicksort não tem a mesma forma para todos os arrays de entrada, dependendo completamente deste. Para o exemplo anterior teremos a seguinte árvore (os nós estão etiquetados com o **comprimento** do vector em cada invocação):



Análise intuitiva

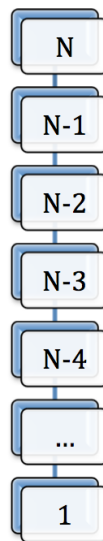
Intuitivamente, o comportamento de pior caso do algoritmo ocorre quando a operação de partição produz o resultado *mais desequilibrado* possível, i.e. quando um dos vectores resultantes é vazio, e o outro contém $N-1$ elementos (todos excepto o pivot). Quando isto acontece em todas as invocações da função de partição, a execução é caracterizada pela seguinte recorrência, em que o termo $\Theta(N)$ corresponde ao tempo de execução da função de partição:

$$T_p(N) = \Theta(1), \text{ se } N \leq 1$$

$$T_p(N) = T_p(N - 1) + \Theta(N), \text{ se } N > 1$$

que tem como solução $T_p(N) = \Theta(N^2)$. Note-se que isto ocorre quando o array se encontra à partida ordenado de forma crescente (ou decrescente)!

[recorde que o algoritmo *insertion sort*, que executa também em tempo quadrático no pior caso, executa em tempo $\Theta(N)$ quando o array de entrada está já ordenado de forma crescente.]



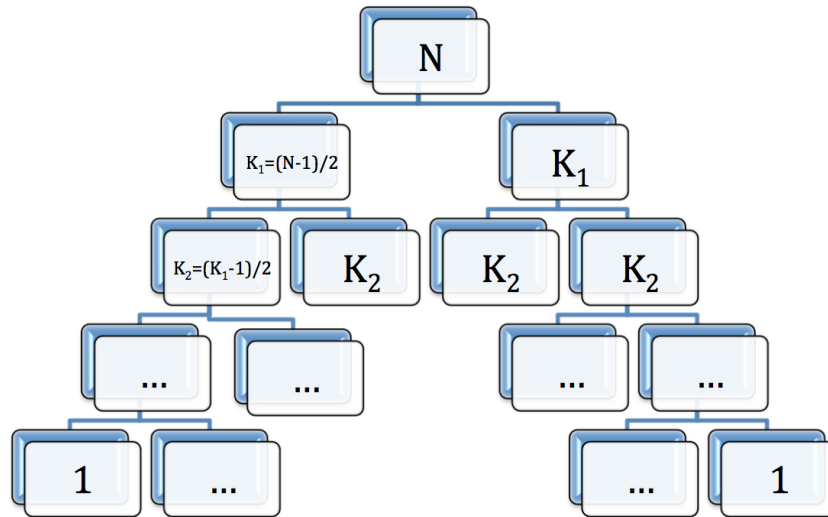
Árvore de recursividade correspondente ao pior caso de quicksort

Quanto ao melhor caso, ele ocorre quando, em todas as execuções da função de partição, ela produz o resultado *mais equilibrado* possível, i.e. quando ambos os vectores resultantes têm comprimento aproximado $\frac{N-1}{2}$. Em termos mais rigorosos a execução do algoritmo será então caracterizada pela seguinte recorrência:

$$T_m(n) = \Theta(n) + T_m(\lfloor n/2 \rfloor) + T_m(\lceil n/2 \rceil - 1)$$

Trata-se de uma recorrência muito semelhante à do algoritmo merge sort, com a mesma solução

$$T_m(N) = \Theta(N \log N).$$



Árvore de recursividade correspondente ao melhor caso de quicksort

Análise de pior caso

Como poderemos obter uma *prova* de que as nossas intuições acima estão de facto correctas? Ou seja, de que de facto o comportamento de *quicksort* é caracterizado por $T(N) = \Omega(N \log N), \mathcal{O}(N^2)$?

Concentremo-nos na análise de pior caso. Observemos antes de mais que podemos descrever o tempo de execução no pior caso de uma forma rigorosa, utilizando para isso um operador de *maximização* sobre a soma do tempo das duas invocações recursivas, em ordem ao comprimento k de um dos vectores resultantes da partição (note-se que o outro vector terá comprimento $N-k-1$).

$$T_p(N) = \Theta(N) + \max_{k=0}^{N-1} (T_p(k) + T_p(N - k - 1))$$

Para mostrarmos que esta recorrência tem a mesma solução que a que escrevemos acima de forma intuitiva, utilizaremos o **método da substituição**.

Admitamos então que $T_p(N) \leq cN^2$ para uma determinada constante c . Então podemos aplicar as seguintes hipóteses de indução:

- $T_p(k) \leq ck^2$
- $T_p(N - k - 1) \leq c(N - k - 1)^2$

e logo,

$$T_p(N) \leq \Theta(N) + \max (ck^2 + c(N - k - 1)^2)$$

$$T_p(N) \leq \Theta(N) + c \max (P(k))$$

$$\text{com } P(k) = k^2 + (N - k - 1)^2 = 2k^2 + (2 - 2N)k + (N - 1)^2$$

Ora, por análise de $P(k)$ conclui-se que os seus máximos no intervalo $0 \leq k \leq N - 1$ se encontram nas extremidades, para $k = 0$ (primeiro vector é vazio) e $k = N - 1$ (segundo vector é vazio).

Para estes valores de k temos $P(0) = P(N - 1) = (N - 1)^2$, logo

$$T_p(N) \leq \Theta(N) + c(N - 1)^2$$

E temos então que $T_p(N) = \Theta(N^2)$, o que conclui a prova pelo método de substituição.

Análise de Caso Médio

O caso médio do tempo de execução pode ser estimado através do valor esperado do número de comparações efectuadas entre elementos do vector:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} P_{comp}(i, j)$$

Admitamos para simplificar que lidamos com sequências que não contêm elementos repetidos.

Começemos por observar que, dados quaisquer dois elementos x e y , eles começam por ser mantidos no mesmo sub-vector, enquanto os pivots forem superiores ou inferiores a ambos.

Até que ocorrerá um de dois cenários:

1. Um dos elementos é usado como *pivot* na partição de um sub-array que contém o outro elemento. É o caso de (2,4) ou (4,7) no exemplo de execução. Neste caso os elementos **são comparados**.
1. Os elementos são *separados* por uma qualquer partição em que é usado um terceiro elemento z como pivot. É o caso do par (2,7) no exemplo.

Neste caso **não são comparados**. Note-se que o pivot z terá de ser um elemento do vector tal que $x < z < y$.

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Conhecendo o conjunto de elementos contidos no array, é possível calcular a probabilidade de qualquer par de elementos (x, y) ser ou não comparado durante a execução do algoritmo: sendo n o número de elementos z tais que $x < z < y$, a probabilidade de x e y serem comparados é $\frac{2}{2+n}$.

No exemplo, a probabilidade de 2 e 7 serem comparados é igual a $2/6 = 1/3$.

Ora, uma forma de conhecermos este número n de elementos contidos entre um par de elementos do array, consiste em observar o array final, ordenado. Se no array ordenado x e y se encontram nas posições i e j , então $n = j - i - 1$, e a probabilidade de terem sido comparados é dada por $\frac{2}{j-i-1+2} = \frac{2}{j-i+1}$.

Basta agora calcular a soma destas probabilidades para todos os pares de elementos:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1}$$

Fazendo uma mudança de variável:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{k=2}^{n-i} \frac{2}{k}$$

$$T_{avg}(n) \leq \sum_{i=0}^{n-2} \sum_{k=2}^n \frac{2}{k} = 2(n-1) \left(\sum_{k=1}^n \frac{1}{k} - 1 \right)$$

Podemos agora utilizar o resultado seguinte:

$$\sum_{k=1}^n \frac{1}{k} \leq 1 + \ln n$$

Para obter

$$T_{avg}(n) = \mathcal{O}(n \lg n)$$

Nesta análise assumimos, naturalmente, que todas as permutações dos elementos da sequência podem ocorrer com igual probabilidade (ou, o que é equivalente, que estamos a analisar uma versão *aleatorizada* do algoritmo).

Algoritmos de ordenação baseados em comparações

Todos os algoritmos estudados até aqui são baseados em **comparações**: dados dois elementos $A[i]$ e $A[j]$, é efectuado um teste (e.g. $A[i] \leq A[j]$) que determina a ordem relativa desses elementos, não sendo usado qualquer outro método para obter informação sobre o valor dos elementos a ordenar.

Admitamos que a sequência não contém elementos repetidos. O **conjunto de execuções** de um algoritmo baseado em comparações (sobre sequências de uma determinada dimensão) pode ser visto de forma abstracta como constituindo uma *Árvore de Decisão*: uma árvore binária cujos **caminhos descendentes**, desde a raiz até às folhas, correspondem às diferentes execuções do algoritmo, como se segue:

- cada nó contém uma *condição*, correspondente a uma **comparação** entre dois elementos, $A[i] \leq A[j]$
- os **caminhos descendentes que chegam** a este nó correspondem às execuções que efectuam esta comparação $A[i] \leq A[j]$
- os caminhos que continuam para a **sub-árvore esquerda** deste nó correspondem às execuções em que o teste $A[i] \leq A[j]$ teve resposta **verdadeira**
- os caminhos que continuam para a **sub-árvore direita** deste nó correspondem às execuções em que o teste $A[i] \leq A[j]$ teve resposta **falsa**

- assim, cada caminho da raiz até uma folha contém **a sequência de comparações efectuadas numa execução** concreta do algoritmo

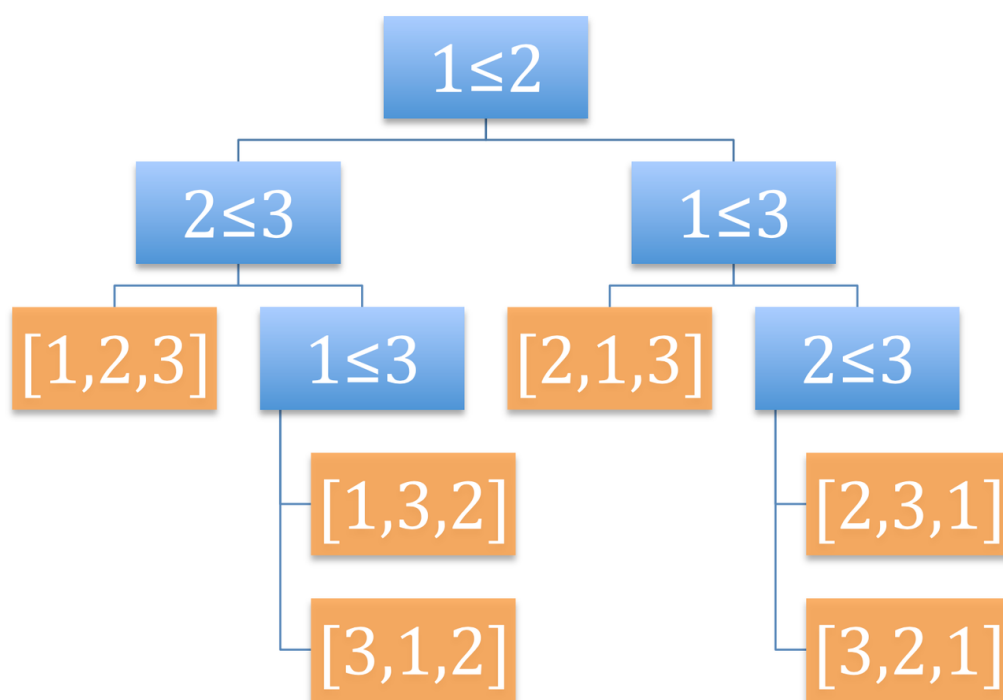
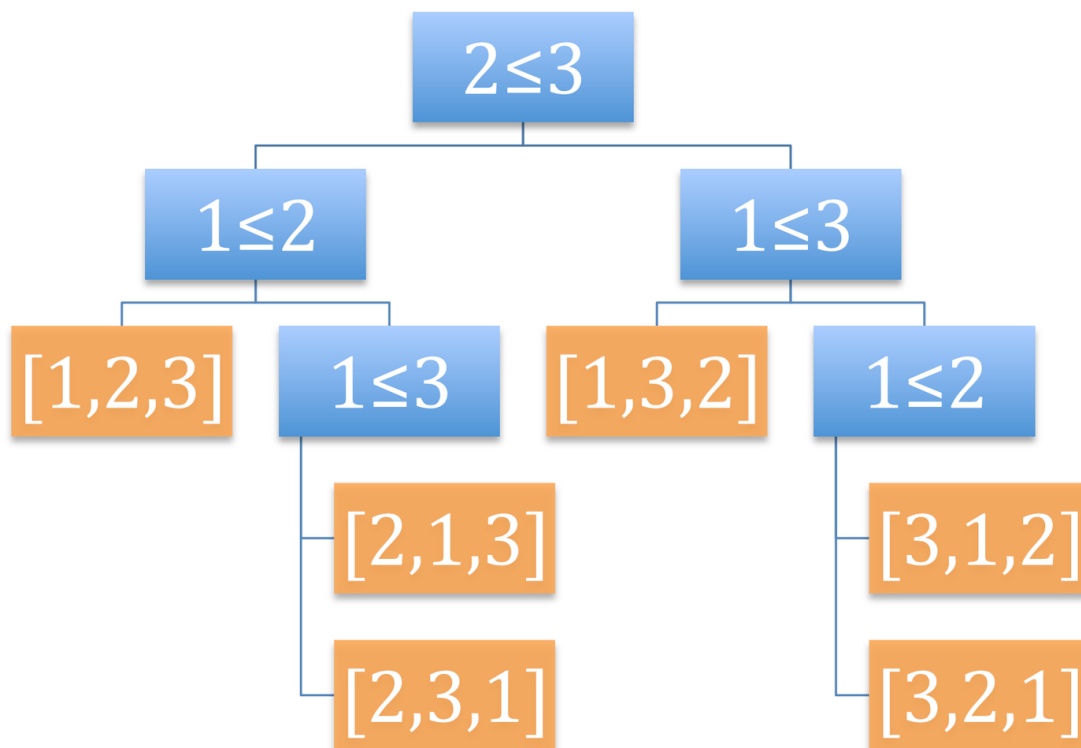
Note-se ainda que:

- Cada folha corresponde a uma ordenação possível do input, ou seja uma permutação possível da sequência inicial
- **Todas** as permutações da sequência devem aparecer como folhas, já que a árvore contempla todas as execuções
- Existem $N!$ permutações

Assim:

*O **número de folhas** da árvore de decisão de um algoritmo de ordenação de um array de comprimento N é igual a $N!$*

Vejamos dois exemplos: a execução dos algoritmos **insertion sort** e **merge sort**, sobre inputs de comprimento 3 (o tamanho das árvores cresce exponencialmente com o comprimento dos array).

Árvore de Decisão insertion sort, $N=3$ Árvore de Decisão merge sort, $N=3$

Observe-se agora que:

- o número de operações de comparação efectuadas numa execução concreta é dado pelo **comprimento do caminho** descendente correspondente a essa execução;
- logo, o número de operações de comparação efectuadas no *pior caso* é dado pela **altura** da árvore de decisão

Teorema

A altura h de uma árvore de decisão tem o seguinte limite mínimo, em que N é o tamanho do input:

$$h \geq \lg(N!)$$

Prova

1. Em geral uma árvore binária de altura h tem **no máximo** 2^h folhas.
2. As árvores que aqui consideramos têm $N!$ folhas, correspondentes a todas as permutações do input
3. Assim, $N! \leq 2^h$
4. Logo, $\lg(N!) \leq h$

Ora, uma vez que num algoritmo de ordenação deste tipo o tempo de execução assintótico pode ser calculado tomando apenas em conta a operação de comparação, temos o seguinte

Corolário

Seja $T(N)$ o tempo de execução no pior caso de um qualquer algoritmo de ordenação baseado em comparações. Então $T(N) = \Omega(N \lg N)$

Prova

Basta usar o seguinte facto: $\lg(N!) = \Theta(N \lg N)$

Conclui-se assim que não é possível bater o comportamento de pior caso do algoritmo *merge sort*: pode-se dizer que é um algoritmo **assintoticamente óptimo** uma vez que o seu tempo de execução no pior caso é $\Theta(N \lg N)$.

Algoritmos *Counting Sort* e *Radix Sort*

Na realidade, é possível ordenar vectores em tempo linear, batendo o limite $N \log N$, se se utilizar um método que não dependa de comparações entre elementos.

O algoritmo *counting sort* pode usado para ordenar sequências de números inteiros, se for conhecida à partida a gama de valores armazenados na sequência. Utiliza para isto um vector auxiliar para armazenar um histograma (uma contagem) dos elementos da sequência a ordenar.

A versão apresentada a seguir deste algoritmo:

- assume que os elementos do array A a ordenar estão contidos no conjunto $\{0 \dots k\}$, sendo o valor de k conhecido
- coloca a sequência ordenada no array B (não é pois um algoritmo de ordenação *in-place*).

```
1 void counting_sort(int A[], int B[], int N, int k) {
2     int C[k+1];
3     for (i=0 ; i<=k ; i++)                /* inicialização de C[]
4         C[i] = 0;
5
6     for (j=0 ; j<N ; j++)                /* contagem ocorr. A[j]
7         C[A[j]] = C[A[j]]+1;
8     for (i=1 ; i<=k ; i++)                /* contagem dos <= i
9         C[i] = C[i]+C[i-1];
10
11     for (j=N-1 ; j>=0 ; j--) {            /* construção do vector ord
12         B[C[A[j]]-1] = A[j];
13         C[A[j]] = C[A[j]]-1;
```

```

14     }
15 }

```

EXEMPLO:

```

1  k = 20
2  A = [10, 5, 20, 10, 17]
3  C = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0] // 1o. ciclo
4  C = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0,
      1] // 2o. ciclo
5  C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4,
      5] // 3o. ciclo
6  B = [--, --, --, --, --]
7  B = [--, --, --, 17, --]
      // 4o. ciclo, 1 it.
8  C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4,
      5]
9  B = [--, --, 10, 17, --]
      // 4o. ciclo, 2 it.
10 C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4,
      5]
11 B = [--, --, 10, 17, 20]
      // 4o. ciclo, 3 it.
12 C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4,
      4]
13 B = [ 5, --, 10, 17, 20]
      // 4o. ciclo, 4 it.
14 C = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4,
      4]
15 B = [ 5, 10, 10, 17, 20]
      // 4o. ciclo, 5 it.
16 C = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 4, 4,
      4]

```

Esta contagem permite colocar (último ciclo for) os elementos de A directamente nas suas posições finais.

A análise do tempo de execução é imediata: $T(N) = \Theta(N + k)$

Se $k = \mathcal{O}(N)$, então $T(N) = \Theta(N)$

Propriedade de Estabilidade

Note-se que o array C:

- começa por guardar uma contagem do número de ocorrências de cada elemento no array A a ordenar
- depois do terceiro ciclo for, passa a guardar em cada posição i uma contagem do **número de elementos inferiores ou iguais a i**

Imaginemos que A contém duas ocorrências de um elemento x , nas posições a e b , sendo $a < b$, e que existem u elementos inferiores a x em A. Teremos então que $C[x] = u + 2$

O algoritmo preencherá o array B percorrendo A do final para o início, e:

1. ao passar pela posição b de A colocará x na posição $u + 2$ e decrementará $C[x]$ para $C[x] = u + 1$
2. mais tarde, ao passar pela posição a de A colocará x na posição $u + 1$ e decrementará $C[x]$ para $C[x] = u$

Sendo assim, o algoritmo counting sort observa a seguinte propriedade de *estabilidade*:

| A ordem das ocorrências em A é preservada em B

Esta propriedade parece inútil quando se considera a ordenação de sequências de números, mas é de facto útil se estes números forem vistos como parte de estruturas contendo outros campos.

Considere-se por exemplo o problema de ordenação de datas. Os 3 campos presentes numa data devem ter prioridades diferentes na ordenação, tendo o *ano* prioridade mais alta, seguindo-se o *mês*, e só depois o *dia*. Um método possível para ordenar datas consiste em fazer ordenações sucessivas usando cada um dos campos, *do menos significativo para o mais significativo*.

Consideremos por exemplo as datas:

- 21/8
- 26/3
- 14/4
- 21/3

Ordenando por dia:

- 14/4
- 21/8
- 21/3
- 26/3

Ordenando agora por mês:

- 21/3
- 26/3
- 14/4
- 21/8

As datas ficam ordenadas, mas é fundamental para isto que ordenação por mês tenha sido estável (foi preservada a ordenação anterior, pelo campo *dia*).

Se acrescentarmos o campo “ano” às datas:

- 21/3/2009
- 26/3/1975
- 14/4/1969
- 21/8/2009

Ordenando por ano é de novo preservada a ordenação anterior:

- 14/4/1969
- 26/3/1975
- 21/3/2009
- 21/8/2009

Mais uma vez a ordenação por ano preservou a ordenação anterior, pelo que as datas ficam ordenadas. Este algoritmo de ordenação de sequências de estruturas multi-campo, recorrendo a ordenações sucessivas usando um algoritmo de ordenação auxiliar estável (por exemplo *counting sort*), é conhecido por **radix sort**.



Criado com o Dropbox Paper. [Saiba mais](#)