ORDENAÇÃO INTERNA DE SEQÜÊNCIAS
DRAFT 0

RAUL H.C.LOPES

1. Introdução

Estas notas introduzem o tópico de ordenação de seqüências, um dos fundamentos da disciplina de *Técnicas de Busca e Ordenação* e do curso de *Ciência da Computação* como um todo. É importante ter ciência de que elas não substituem um bom texto de referência sobre o assunto. O texto clássico de Knuth, [4], é a referência fundamental sobre ordenação; [7] e [3] podem servir como textos auxiliares (mais fáceis de ler, mas menos completos.) Em adição, [8] é introdução excelente e obrigatória à análise de algoritmos, contendo no capítulo inicial estudo detalhado de *quicksort* e *mergesort.*

2. Medidas de complexidade

Complexidade concreta de algoritmos[1] refere-se à avaliação da quantidade de recuros demandados por um algoritmo para resolver um problema. Por recursos gastos pode-se entender:

- número de processadores;
- memória;
- tempo.

Qualquer dessas medidas pode se revelar inadequada se adotada de forma estrita. Por exemplo, memória poderia ser medida em palavras, mas o tamanho da palavra dos computadores varia com o fabricante e a geração do processador. O mesmo vale para tempo: avaliações de tempo em relógio são inadequadas porque o número de instruções executadas em um segundo por um processador muda todos os anos. Comparações de desempenho de programas só fazem sentido em experimentos em que o ambiente, incluindo máquina e dados usados, estão muito bem definidos e controlados e, na verdade, só fazem sentido quando o estado da arte da *Análise de Algoritmos* não permite análise mais

[1]Termo introduzido por Knuth, veja artigo *Mathematical Analysis of Algorithms* em [5].

```
seqsum [] = 0
seqsum (x:xs) = x+seqsum xs
```

FIGURA 1. Soma de elementos de uma seqüência

detalhada dos algoritmos em estudo. [2] Consequentemente, medidas mais abstratas de complexidade são usadas. Por exemplo, para avaliar a complexidade de tempo de algoritmo é usual considerar: número de transições executadas, número de chamadas recursivas, número de comparaçõesou operações aritméticas, etc.

Por outro lado, é importante considerar, estando definida, por exemplo, a medida de tempo usar, que um algoritmo A pode demandar tempos complemente diferentes quando usado para conjuntos de dados diferentes, mesmo que esses conjuntos tenham o mesmo número de elementos. Um algoritmo pode, por exemplo, demandar mil comparações de itens para ordenar um conjunto de mil inteiros e demandar um milhão de comparações para um outro conjunto de mil inteiros. A análise de complexidade lida ao menos com a avaliação dos seguintes casos:

- **melhor caso:** qual é o melhor desempenho do algoritmo para um conjunto de dados de tamanho n e que características desse conjunto determinam tal desempenho;
- **pior desempenho:** qual é o pior desempenho possível do algoritmo.
- **caso médio:** qual é o desempenho médio quando consideradas todas as distribuições *prováveis* dos possíveis conjuntos de dados a usar.

Considere o programa em Haskell da figura 1.

Se medirmos o tempo desse programa por chamadas recursivas para uma entrada com n itens obtemos as equações:

$$T(0) = 0T(n+1) = 1 + T(n)$$

[2]Na maioria dos casos, estudos de programas para avaliação de desempenho revelam incompetência inerente do *experimentador*.

Parece óbvio verificar que esse sistema de equações[3] na verdade representa a somatória

$$\sum_{1 \leq i \leq n} 1$$

dando o valor de $T(n) = n$.

Exercício 1. *Verifica o resultado da recorrência usando indução.*

Note também que:

- Esse algoritmo executa a cada chamada recursiva: uma adição, uma decomposição de lista (em *head* e *tail*) e uma comparação, para deteminar se a lista é vazia.
- Esse algoritmo executa, na última chamada recursiva, apenas uma comparação para detectar se a seqüência é vazia.

Somando tudo sentimo-nos tentados a dizer que o número total de operações do algoritmo é: $4n + 1$. O problema é que, dependendo da tecnologia do processador e do compilador cada uma dessas operações poderá ser decomposta em várias instruções. Por causa disso, é mais adequado dizer que esse algoritmo demanada kn operações para alguma constante k que será pequena quando o n crescer arbitrariamente. Mais exatamente, diz-se que

$$T(n) \in \Theta(n)$$

E o que é $\Theta(n)$?

Em 1976, Knuth propôs [4] a seguinte notação para descrever o comportamento assintótico de funções:

- $O(f(n))$ denota o conjunto de todas as funções $g(n)$ tal que existem constantes positivas C e n_0 tal que

$$\forall n : n \geq n_0 : |g(n)| \leq Cf(n)$$

- $\Omega(f(n))$ denota o conjunto de todas as funções $g(n)$ tal que existem constantes positivas C e n_0 tal que

$$\forall n : n \geq n_0 : g(n) \geq Cf(n)$$

- $\Theta(f(n))$ denota o conjunto de todas as funções $g(n)$ tal que existem constantes positivas C, C' e n_0 tal que

$$\forall n : n \geq n_0 : Cf(n) \leq g(n) \leq Cf(n)$$

[3] Chamado recorrente por causa da ocorrência de $T(n)$ na equação que dá o valor de $T(n + 1)$.

[4] Artigo intitulado *Big Omicron and Big Omega and Big Theta* publicado em na revista SIGACT News e na coletânea [5]. Esse artigo continua sendo a explanação mais clara do que seja essa notação e de sua origem. Deveria ser leitua obrigatória de todos os alunos de *Ciência de Computação.*

```
linearSearch x [] = False
linearSearch x (y:ys)
          | x == y = True
          | otherwise = linearSearch x ys
```

FIGURA 2. Pesquisa linear

Exercício 2. *Apresenta definição alternativa para essas três noatções baseada em estudo de crescimento de funções.*

A figura 2 apresenta programa em Haskell para fazer uma busca linear em uma seqüência por um elemento dado.

Dada uma seqüência de n elementos e um elemento x esse algoritmo pode:

- encontrar o elemento procurado na primeira posição da seqüência: isso demandaria o teste para determinar que a seqüência não é vazia e o teste para determinar que x e o primeiro elemento da seqüência são iguais;
- comparar x com cada elemento e descobrir que ele não ocorre na seqüência: isso demandaria n chamadas recursivas, $n + 1$ testes de seqüência vazia, n comparações de itens e n operações de desconstrução da seqüência.

Num caso desses afirma-se que

- no melhor caso $T(n) \in \Omega(1)$: porque o menor tempo de execução desse algritmo é constante.
- no pior caso $T(n) \in O(n)$: porque o pior tempo do algoritmo é menor ou igual a kn para alguma constante k.

Exercício 3. *Em cada caso, determine a veracidade (e justifique):*

- $n \in O(n^2)$
- $n^2 in \Omega(n)$
- $n \lg n + n \in O(n \log n)$
- $n^2 + n \in O(n^2)$

3. Ordenação por comparação

Nas próximas seções assuma que o objetivo consiste em ordenar seqüências de itens e que essas seqüências estão representadas como vetores. Em princípio todas as seqüências contêm inteiros.

Valem as seguintes definições[5]:

Definição 1.

$$S(0..m) \nearrow \stackrel{\triangle}{=} \forall i : 0 \leq i < j < m : S.i \leq S.j \tag{1}$$

$$S(0..m) \searrow \stackrel{\triangle}{=} \forall i : 0 \leq i < j < m : S.i \geq S.j \tag{2}$$

$$S(0..m) \uparrow \stackrel{\triangle}{=} \forall i : 0 \leq i < j < m : S.i < S.j \tag{3}$$

$$S(0..m) \downarrow \stackrel{\triangle}{=} \forall i : 0 \leq i < j < m : S.i > S.j \tag{4}$$

Os métodos de ordenação de seqüências são geralmente classificados em dois tipos:

- ordenação por comparação: o passo fundamental da ordenação consiste na comparação entre itens do conjunto a ordenar.
- ordenação por distribuição: não existem comparação entre itens e o passo fundamental da ordenação é a separação dos itens em grupos pela raiz do alfabeto usado para construí-los.

Teorema 1. *Uma árvore binária de altura h tem no máximo 2^h folhas.*

Prova.

Exercício.

□

Um algoritmo para ordenar uma seqüência de n elemtentos distintos via comparação de chaves constrói uma árvore binária de decisão onde cada nó representa uma comparação entre dois elementos a e b: o filho da esquerda desse nó armazena a seqüência de decisões do algoritmo quando $a < b$, sendo armazenada no filho da direita a seqüência de decisões adotadas quando $a > b$.

Teorema 2. *Uma árvore decisão para ordenar n elementos tem altura maior ou igual a $\lg n$.*

Prova.

Exercício.

□

Teorema 3. *Um algoritmo de ordenação por comparação demanda $\Omega n \log n$ comparações de elementos.*

[5] A notação é emprestada de [6]

4. Ordenação por inserção

O processo de ordenação por inserção consiste em repetidamente inserir elementos da seqüência em questão em uma subseqüência previamente ordenada. Isso dá basicamente o mesmo *template* que a soma de elementos de uma seqüência:

- para somar n elementos, acumule os primeiros $n-1$ e some o resultado com o elemento restante.
- para ordenar uma seqüência de n elementos, ordene os primeiros $n-1$ e insira em ordem na subseqüência resultante o elemento restante.

Essa observação leva à invariante do loop principal:

$$P \triangleq 0 \leq i \leq m \wedge S(0..i) \nearrow$$

Os elementos restantes da derivação são óbvios:

- variável derivada: $T \triangleq m - i$, que o número de elementos ainda não inseridos;
- a guarda do loop: $B \triangleq m - i > 0$, que estabelece o limite da computação.
- a condição inicial: $i = 1$, assumindo que a subseqüência com o primeiro elemento está ordenada. Note que mesmo que a seqüência seja vazia, esta inicialização ainda é consistente com a guarda do loop.
- a transição do loop, realizada até o ponto-fixo, deve inserir novo elemento na porção ordenada e restabelecer a invariante:

```
insert(S,i)
; i:=i+1
```

Ao final do algoritmo, a condição R estabelecida deve ser a mesma de qualquer algoritmo de ordenação:

- $S(0..m) \nearrow$
- S é uma permutação da seqüência inicial.

Para derivar o algoritmo inserção de um elemento na porção ordenada, leve em conta:

- a situação inicial: $S(0..i) \nearrow$
- que se $S.i$ for preservado em alguma variaável temporária tmp e encontrada a posição correta para ele, a subseqüência poderá ter o seguinte estado antes da inserção:
  - trecho inicial não alterado: $S(0..j) \nearrow$
  - buraco aberto para inserção de tmp em j
  - trecho que foi deslocado para a direita: $S(j+1..i+1) \nearrow$

```
insertionsort(S,m) is
⟦ Const B.0..m−1 ∈ Int;
    Var i ∈ Int;
    initially B=S ∧ i=0 ∧ m≥0
    {invariant P}
    {bound T}
    do
      G: m−i > 0: insert(S,i)
                        ;i:=i+1
    od
    {P ∧ ¬B}

  {R}
⟧


insert(S,i) is
⟦ Var j,tmp ∈ Int;
    initially j=i ∧ tmp=S.i
    do
       j > 0: S.j,j:=S.j−1,j−1
    od
⟧
```

FIGURA 3. Ordenação por inserção

Exercício 4. *Prove a correção do algoritmo da figura 3.*

4.1. **Complexidade de *insert*.** O trecho que insere um novo elemento em uma subseqüência de i itens previamente ordenada pode realizar:

- uma única comparação: caso em que o novo item é maior que todos os outros presentes na subseqüência;
- comparar o novo item com todos os outros e movimentar cada um deles para a direita, produzindo: m comparações de itens, $i+1$ movimentos de itens, i transições.

4.2. **Complexidade do *insertion sort*.** O algoritmo *insertion sort* chama m para ordenar uma seqüência de m elementos.

Exercício 5. *Prove que se $C(n)$ e $M(n)$ são respectivamente número de comparações e movimentos para ordenar uma seqüência de m itens usando* insertion sort*, então:*

$$C(m) \in O(m^2)$$
$$M(m) \in O(m^2)$$

O *lower bound* para resolver uma classe de problemas P

Definição 2. *Um algoritmo A para resolver uma classe de problemas P é ótimo em relação a um critério de complexidade $C \triangleq$ A resolve qualquer problema da classe P de tamanho m com complexidade igual ao lower bound da classe P.*

Exercício 6. *Prove que uma*

¡++¿

5. Ordenação por seleção

Algoritmos de ordenação por seleção, da mesma forma que o *insertion sort*, assume uma seção do vetor ordenada e a ele acresentam, a cada transição o novo elemento: desta vez, o maior elemento presente na seção não ordenada. Como antes o objetivo é estabelecer:

$$R \triangleq \ S(0..m) \nearrow \wedge S \in \pi.B$$

onde B é o valor inicial do vetor a ordenar e $\pi.B$ é o conjunto de todas as permutações de B.

Nos algoritmos de ordenação por seleção generaliza-se a propriedade R, para obter a invariante ou parte dela, trocando, por exemplo, a constante 0 por uma variável i.

$$P_0 \triangleq \ 0 \leq i \leq m \wedge S(i+1..m) \nearrow \wedge S \in \pi.B$$

Dado que P_0 estabelece que a seção $S(i+1..m-1)$ está ordenada, é razoável assumir que essa seção não será mais alterada e que outros elementos presentes em S são menores do $S(i+1)$. Raciocínio análogo seria usado para construir a ordenação se a hipótese fosse de a seção ordenada de S é inicial e que os elementos que não estão nessa seção são maiores do que os que nela estão presentes.

5.1. Em tempo quadrático: *selection sort*.

No algoritmo *selection sort*, o novo elemento adicionado a cada passo é obtido pela escolha do maior elemento presente na seção não ordenada. A invariante é:

$$P \triangleq \ 0 \leq i < m \wedge S(i+1..m) \nearrow \wedge S(i+1) > S(0..i) \wedge S \in \pi.B$$

```
selectionsort(S,m) is
{S(0..m − 1) ∈ Int ∧ m > 0}
⟦ Var i ∈ Int;
    initially B=S ∧ i=m−1
    do {P}
       i > 0: maxpos(S,i,j)
              ;S.j,S.i,i:=S.i,S.j,i−1
    od
    {P ∧ ¬G}

  {R}
⟧
```

FIGURA 4. Algoritmo *selection sort*

A varaiável derivada é T

$$T \triangleq i$$

A guarda do loop estabelece que, quando só existe um elemento na seção não ordenada, o loop atinge seu ponto-fixo: uma seção de um elemento está ordenada.

$$G \triangleq i > 0$$

O algoritmo está na figura 4. Nele **maxpos** estabelece a posição de *supremum* para a seção não ordenada. Esse procedimento poderia ser definido como:

$$maxpos(S, i) \triangleq j :- \forall k : 0 \leq k \leq i : S.j \geq S.k$$

A definição determinística de *maxpos* está na figura 5.

Teorema 4. *Prove que maxpos definido na figura 5 calcula um j tal que $0 \leq j \leq i$ e $S.j \geq S(0..i)$.*

5.1.1. *A complexidade de* maxpos. O algoritmo da figura 5 não movimenta nenhum elemento e, na verdade, no caso de a seqüência já estar ordenada, *selection sort* não realiza movimentos de chaves. Por outro lado, *maxpos* sempre realiza $i-1$ comparações: *maxpos* vê cada seção que recebe como desconhecida e, por isso, precisa comparar seu último elemento com todos os outros para obter o supremum da seção.

```
maxpos(S,i) is
[[ Var k ∈ Int;
   initially k=i-1 ∧ j=i
   do
     k≥0 → if
                S.k > S.j → j,k:=k,k-1
             [] S.k≤S.j → k:= k-1
           fi
   od
]]
```

FIGURA 5. Algoritmo para seleção de máximo

5.1.2. *Complexidade do* selection sort. Cada transição do algoritmo da figura 4 realiza, no máximo, uma troca de chaves: o que daria dois movimentos de chaves. Logo, o algoritmo realiza de zero a $2m$ movimentos de chaves.

O número de comparações realizadas é dado por $C(m)$ abaixo:

$$C(m) = (+i : 1 \leq i \leq m : i \in \Theta(m^2)$$

Em resumo, independentemente da seqüência dado, *selection sort* sempre demanda m^2 transições e comparações, onde m é o número de itens na seqüência. Isso está muito longe do *lower-bound* possível para ordenação por comparação que demanda $\Omega(m \lg m)$ comparações.

5.2. **Em tempo ótimo: *heapsort*.** O problema essencial do método *selection sort* parece ser o fato de que cada transição vê a seção do vetor como totalmente nova: nenhum cohnecimento é trazido da transição anterior. Para melhorar o desempenho do método será necessário considerar a possibilidade de:

- manter a seção a ser pesquisada, na busca de elemeto máximo, em uma estrutura que permite que a seleção seja feita em tempo constante: se não for constante, crescerá com o tamanho da seção e o comportamento quadrático reaparecerá;
- garantir que a estrutura possa ter suas propriedades invariantes rapidamente restauradas após a operação de extração de um máximo;
- dado que se objetiva um tempo $O(m \lg m)$ para o processo completo de ordenação e que o *template* geral do *selection sort* será mantido, esta é uma tentativa de melhorar seu desempenho e não de descartálo, cada transição, composta pela extração do

máximo e ajuste da estrura às suas propriedades invariantes, deveria demandar $O(\log i)$ operações: isso garantiria um total de operações de

$$(+i : 1 \leq i \leq m : \lg i) \in O(m \lg m)$$

Mesmo sem considerar concretamente nenhuma estrutura, dá para antecipar uma invariante para o novo algoritmo:

$$logstruct.S, 0, i \wedge S(i+1..m) \nearrow \wedge S \in \pi.B$$

onde a propriedade *logstruct* é a invariante de uma estrutua que:

- permite extração de um máximo em $O(1)$;
- permite ajuste á invariante em $O(\lg i)$.

A estrutura a ser usada é um heap e o algoritmo será por isso denominado heapsort. O heap terá a seguinte propriedade invariante:

$$\begin{aligned} heap.S, i, m \triangleq \quad & \forall j : i \leq j \leq m \wedge 2j < m : S.j \geq S.2j \\ & \forall j : i \leq j \leq m \wedge 2j+1 < m : S.j \geq S(2j+1) \end{aligned}$$

Um heap é uma árvore binária balanceada, qualquer *path* da raiz a uma folha tem comprimento $O(\lg m)$ em que a raiz de qualquer subárvore é maior ou igual do que qualquer outro elemento dessa subárvore: isso garante localização e extração do máximo em $O(1)$, afinal ele estará localizado na raiz do heap. A acomodação do heap em um array, importante para garantir que o processo de ordenação ocorre no próprio array dado, consiste em representar os dois filhos de uma subárvore com raiz na posição i como árvores, heaps na verdade, com raízes em $2i$ e $2i+1$.

A invariante do *heapsort* é a propriedade P

$$P \triangleq \ 0 \leq i < m \wedge heap.S, 0, i \wedge S(i+1..m) \nearrow \wedge S \in \pi.B$$

O algoritmo *heapsink* é responsável por restaurar o heap após a extração do elemento da raiz: após a troca que leva o máximo para a posição final do heap, existe um elemento a menos na estrutura e o elemento que se encontra na raiz está fora de ordem. O algoritmo está na figura 7.

Exercício 7. *Prove que* heapsink *restaura a propriedade de heap na seção não ordenada do array.*

Exercício 8. *Prove que se o heap contém i elementos, então* heapsink *demanda $O(\lg i)$ transições.*

O heap inicial é construído pelo algoritmo *heapbuild*, na figura 8.

```
heapsort(S,m) is
{S(0..m − 1) ∈ Int ∧ m > 0}
[[ Var i ∈ Int;
    initially B=S ∧ i=m−1
    heapbuild(S,m)
   ;do {P}
      i>0 → S.j,S.i,i:=S.i,S.j,i−1
            ;heapsink(S,i)
    od
    {P ∧ ¬G}
    {R}
]]
```

FIGURA 6. Algoritmo *heapsort*

```
heapsink(S,i,n) is
{i > 0 ∧ n > 0}
[[ Var j ∈ Int;
   initially j=i
   ;do
      2j<n → let
                k:= 2j+1, if (2j+1)<n
                             cand S(2j+1)> S.2j
                          2j, otherwise
             in
                if
                  S.j<S.k → S.j,S.k,j:=S.k,S.j,k
                 [] S.j≥S.k → j:= n
                fi
             tel
    od
]]
```

FIGURA 7. Algoritmo *heapsink*

Exercício 9. *Prove que* heapsink *demanda* $O(\lg(m-i))$ *transições, comparações e movimentos de chaves*

```
heapbuild(S,m) is
{S(0..m − 1) ∈ Int ∧ m > 0}
⟦ Var j ∈ Int;
  initially j=m−2
  do
     m≥0 → heapsink(S, j ,m)
           ; j:=j−1
  od
⟧
```

FIGURA 8. Algoritmo *heapbuild*

Exercício 10. *Prove que* heapbuild *demanda*

$$(+i : 1 \leq i < m : \lg(m - i)) = \lg(m!) \in O(m \lg m)$$

transições.

Exercício 11. *Prove que o* heapsort *demanda*

$$(\lg m!) + (+i : 1 \leq i < m : \lg(m - i)) \in O(m \lg m)$$

transições, movimentos e comparações de chaves.

Exercício 12. *Prove o* heapsort *é um algoritmo ótimo para o problema de ordenação baseado em comparação.*

6. Ordenação por troca

Métodos de ordenação por troca repetem a troca pares de elementos que se encontram fora de ordem até obter a ordem total da seqüência. O mais notável desses algoritmos é o *quicksort*, cujo algoritmo espelha a construção de uma árvore binária para os elementos da seqüência em questão:

- escolha arbitrária de um *pivô* para a raiz da árvore;
- particionamento do conjunto restante em elementos menores ou iguais ao pivô e elementos maiores ou iguais ao pivô;
- repetição do processo para ordenação de cada uma das partições.

É um algoritmo típico do paradigma *dividir para conquistar*: para resolver um problema (no caso, ordenar uma seqüência) divida-o em subproblemas e resolva-os pelo mesmo processo (a ordenação das partições). Finalmente, combine as soluções parciais: no caso concatenar as subseqüências ordenadas. A figura 9 apresenta um algoritmo de *quicksort* escrito em Haskell.

```
module Quicksort where

partition [] = ([],[],[])
partition (x:xs) = foldl (partadd x) ([],[],[]) (x:xs)

partadd x (xs,ys,zs) u
  | u < x  = (u:xs,ys,zs)
  | u == x = (xs,u:ys,zs)
  | u > x  = (xs,ys,u:zs)

quicksort [] = []
quicksort [x] = [x]
quicksort xs = (quicksort lt)++eqs++(quicksort gt)
   where (lt,eqs,gt) = partition3way xs
```

FIGURA 9. Algoritmo *quicksort*

Note que a função *partition* particiona uma seqüência dada em três subseqüências: de elementos menores, iguais e maiores que o pivô, respectivamente. Note ainda que o pivô é sempre o primeiro elemento da seqüência a ordenar.

O algoritmo em Haskell é simples e está expresso em 15 linhas. Se a função *parttion* da biblioteca *Lists* tivesse sido usada esse algoritmo poderia ter uma cinco linhas. No entanto, ele exibe ao menos uma caracterís tica desagradável: cada chamada recursiva demanda criação de três novas seqüências.

Exercício 13. *Prove que para ordenar uma seqüência de m elementos esse algoritmo pode demandar a criação $O(m^2)$ listas.*

Exercício 14. *Prove que os algoritmo* insertion sort*,* selection sort*e* heapsort*, anteriormente apresentados usam espaço $O(1)$ além do espaço do vetor inicial.*

Para evitar a construção de novas listas a cada particionamento, a solução é reorganizar a própria seqüência de entrada que deverá estar representada em um vetor. O algoritmo de particioanemto de Lomut[6] particiona uma seqüência dada, usandp o elemento inicial como pivô em duas seqüência de elementos menores ou iguas e maiores ou iguais ao pivô, respectivamente.

[6]Apresentado nas aulas de aula sobre *"Correçãode Programas."*

Exercício 15. *Prove que o algoritmo de Lomut demanda $O(m)$ operações, o que inclui comparaçõesde chaves, movimentos, transições e quaiquer outras operações durante o processo de ordenação.*

Podemos entender a oordenação via *quicksort* como a realização de tarefas orientada por uma agenda:

- cada tarefa na agenda indica uma partição (seção do vetor dado) a ordenar;
- partições (seções do vetor) ausentes da agenda estão ordenadas.

Nesse contexto, cada tarefa na agenda poderia ser representada por um par definindo os limites inferior e superior da seção a ordenar. A genda em si poderia ser uma seqüência, representada como uma fila ou um stack, por exemplo. Se representada como um stack, ela poderia estar armazenada no próprio stack do programa, o que implica em codificar o *quicksort* como algoritmo recursivo, que é alternativa usada [3], capítulo 7, [2], coluna 11, e [1]. A coluna 11 de [2] apresenta um estudo detalhado, em estilo bastante informal, embora preciso, de variações sobre algoritmos de particionamento. Entre elas, encontra-se o algoritmo de Bentley para particionamento em três vias de Bentley, que é usado em [1] em algoritmo de *quicksort* ternário para ordenação de strings.

Exercício 16. *Detalhe o algoritmo de particionamento de três vias Bentley na linguagem de comandos guardados e prove sua correção.*

Exercício 17. *Prove que no melhor caso o algoritmo de* quicksort *demanda $Theta(m \lg m)$ comparações e movimentos de dados.*

Exercício 18. *Prove que o algoritmo de* quicksort *demanda $O(m^2)$ comparações de chaves.*

7. Ordenação por intercalação

8. Ordenação por distribuição

Referências

1. John Bentley and Robert Sedgewick, *Ternary tree*, Dr. Dobb's (1998).
2. Jon Bentley, *Programming pearls*, second ed., Addison-Wesley, 2000.
3. Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, and Clifford Stein, *Introduction to algorithms*, MIT Press, 2001.
4. Donald E. Knuth, *The art of computer programmimg, volume 3: Sorting and searching*, Addison-Wesley, 1998.
5. _____, *Selected papers on analysis of algorithms*, CSLI, 2000.
6. Tom M.Apostol, *Calculus: One-variable calculus, with an introduction to linear algebra*, second edition ed., vol. I, John Wiley & Sons, Inc., 1967.
7. Robert Sedgewick, *Algorithms*, Addison-Wesley, 1988.

8. Robert Sedgewick and Philip Flajolet, *Introduction to the analysis of algorithms*, Addison-Wesley, 1996.