
QUICKSORT (Cormen, cap. 7)

O algoritmo abaixo resolve o problema de ordenação por meio da estratégia de divisão e conquista. Porém, diferentemente da ordenação por entrelaçamento (Mergesort), aqui o passo mais relevante é o de divisão, no qual o vetor é particionado entre os elementos menores ou iguais ao pivô e os elementos maiores que o pivô (ficando o pivô entre as duas seções). Além disso, o passo de combinação das soluções é aqui trivial, pois, estando o vetor particionado e as duas partições ordenadas, o vetor completo está então ordenado.

```
==========================================================================
Algoritmo:    quicksort
Entrada:      vetor V[1..n] e índices "i" e "f".
Pré-condição: se i < f, então 1 <= i < f <= n.
Saída:        nenhuma.
Pós-condição: se i <= f e n != 0, então V[i..f] estará ordenado.
--------------------------------------------------------------------------
1. SE i < f
2. | p := escolher_pivo(V,i,f)
3. | p := particionar(V,i,p,f)
4. | quicksort(V,i,p-1)
5. | quicksort(V,p+1,f)
==========================================================================
```

O cerne do algoritmo acima é a função "particionar", que pode ser implementada assim:

```
  ============================================================================
  Algoritmo:    particionar_hoare
  Entrada:      vetor V[1..n] e índices "i", "p" e "f".
  Pré-condição: 1 <= i <= p <= f <= n.
  Saída:        um índice p'.
  Pós-condição: (1) Os elementos de "V" ao final serão uma permutação dos
                    elementos de "V" no início da chamada.
                (2) O elemento V[p'] do final da chamada é o elemento V[p] do
                    início da chamada.
                (3) Ao final, para todo i <= j <= f, temos:
                    * j < p' -> V[j] <= V[p'];
                    * j > p' -> V[j] >  V[p'].
  ----------------------------------------------------------------------------
  01. SE i = f
  02. | RETORNE i.
  03. pivo := V[p], V[p] := V[i], V[i] := pivo
  04. a := i+1, b := f
      /*
       *  INVARIANTES:  * V[i] = pivo
       *                * ∀ i < j <  a, V[j] <= pivo.
       *                * ∀ b < j <= f, V[j] >  pivo.
       *                * i < a <= b+1 <= f+1
       *
       *  VARIANTE: b+1-a
       */
  05. REPITA
  06. | ENQUANTO V[a] <= pivo
  07. | | ++a
  08. | | SE b < a
  09. | | | VA_PARA a_passou_b.  // "goto"
  10. | ENQUANTO V[b] > pivo
  11. | | --b
  12. | | SE b < a
  13. | | | VA_PARA a_passou_b.  // "goto"
  14. | aux := V[b], V[b] := V[a], V[a] := aux
  15. | ++a, --b
  16. ENQUANTO a <= b.
  17. RÓTULO a_passou_b:
  18. aux := V[b], V[b] := V[i], V[i] := aux
  19. RETORNE b.
  ============================================================================
```

EXERCÍCIOS E LEITURA ADICIONAL

Exercícios:

a) Prove que toda chamada quicksort(V,i,f) executa em $O(t^2)$, sendo $t = j-i+1$ se $i <= j$, e $t = 0$ se $j < i$.

b) PARTIÇÃO DE LOMUTO: escreva um algoritmo de partição que, ao invés de manter uma partição da forma (onde "(x)" denota o pivô)

```
|(x)| <= x |  ?  | > x |
```

em cada iteração, mantém uma partição da forma

```
|(x)| <= x | > x |  ?  | .
```

Nesse caso, o vetor é percorrido apenas da esquerda para a direita.
(A resposta deste exercício se encontra abaixo, mas tente responder por conta própria antes de olhá-la.)

c) TRIPLA PARTIÇÃO: escreva uma variação do algoritmo do item anterior que, em cada iteração, mantenha uma partição da forma

```
| < x | = x | > x |  ?  | .
```

Ao final, devem ser retornados DOIS ÍNDICES, indicando o início e o fim da seção de elementos iguais ao pivô.

O seguinte é um excelente relato sobre otimização do Quicksort na prática, incluindo as otimizações da função "qsort" da biblioteca padrão da linguagem C:

http://dx.doi.org/10.1002%2Fspe.4380231105

PARTIÇÃO DE LOMUTO (RESPOSTA AO EXERCÍCIO ACIMA)

Aqui está um algoritmo que implementa a ideia de partição de Lomuto:

```
================================================================================
Algoritmo:    particionar_lomuto
Entrada:      vetor V[1..n] e índices "i", "p" e "f".
Pré-condição: 1 <= i <= p <= f <= n.
Saída:        um índice p'.
Pós-condição: (1) Os elementos de "V" ao final serão uma permutação dos
                  elementos de "V" no início da chamada.
              (2) O elemento V[p'] do final da chamada é o elemento V[p] do
                  início da chamada.
              (3) Ao final, para todo i <= j <= f, temos:
                  * j < p' -> V[j] <= V[p'];
                  * j > p' -> V[j] >  V[p'].
--------------------------------------------------------------------------------
01. SE i = f
02. | RETORNE i.
03. pivo := V[p], V[p] := V[i], V[i] := pivo
04. me := i  // Último "Menor ou Igual"
05. PARA j DE i+1 A f
06. | SE V[j] <= pivo
07. | | ++me
08. | | aux := V[j], V[j] := V[me], V[me] := aux
09. aux := V[me], V[me] := V[i], V[i] := aux
10. RETORNE me.
================================================================================
```