
ORDENAÇÃO SEM COMPARAÇÕES (Cormen, cap. 8)

Como mostra o teorema 2 abaixo, existe um limite inferior para o pior caso do tempo de execução de qualquer algoritmo de ordenação baseado exclusivamente em comparações:

1. LEMA: lg(n!) = θ(n*lg n).

==========
PROVA:

Nós mostraremos primeiramente que lg(n!) = O(n*lg n), e depois que
lg(n!) = Ω(n*lg n).

Para mostrar que lg(n!) = O(n * lg n), nós mostraremos o resultado mais forte de que lg(n!) <= n*lg(n), para qualquer natural n > 0. De fato, para n > 0, temos:

```
lg(n!)  = lg(1  *    2  * ... *    n)
        = lg(1) + lg(2) + ... + lg(n)
       <= lg(n) + lg(n) + ... + lg(n)
       <= n*lg(n), CQD.
```

Resta agora mostrar que lg(n!) = Ω(n*lg n), isto é, que existem um real c e um natural n_0 positivos tais que 0 <= c*n*lg(n) <= lg(n!) para todo n >= n_0. Sejam então c = 1/2 e n_0 = 1. Como 0 <= n*lg(n)/2 para todo n >= 1, resta então mostrar que n*lg(n) <= 2*lg(n!) para todo n >= 1. De fato, seja n >= 1. Temos:

```
   n*lg(n) <= 2*lg(n!)

<=> n*lg(n) <= lg(1*2*...*n) + lg(1*2*...*n)

<=> lg(n) + lg(n)   + lg(n)   + ... + lg(n)   + lg(n)   + lg(n) <=
    lg(1) + lg(2)   + lg(3)   + ... + lg(n-2) + lg(n-1) + lg(n)
    lg(n) + lg(n-1) + lg(n-2) + ... + lg(3)   + lg(2)   + lg(1).
```

Observe que, para mostrar a última inequação acima, é suficiente mostrar que, para todo i de 1 a n, temos

lg(n) <= lg(i) + lg(n-i+1).

De fato, para i = 1 e i = n, o resultado é imediato, pois

lg(n) <= 0 + lg(n) = lg(1) + lg(n), CQD.

Seja agora i de 2 a n-1. Temos então:

```
    lg(n) <= lg(i) + lg(n-i+1)
<=> lg(n) <= lg(i*(n-i+1))
<=> n <= i*(n -i +1)
<=> n <= in -i² +i
<=> i² -i <= in -n
<=> i(i-1) <= n(i-1)
<=> i <= n, o que é verdade, CQD.  --> Observe que i-1 != 0, já que i >= 2.
```

==========

A demonstração abaixo usa o conceito de "árvore de decisão", que você pode consultar em:

  * Cormen, §8.1.
  * http://lia.ufc.br/~rudini/ufc/cana.ordinf.pdf
  * http://en.wikipedia.org/wiki/Decision_tree

    (veja também http://en.wikipedia.org/wiki/Comparison_sort).

2. TEOREMA: todo algoritmo de ordenação correto e baseado exclusivamente em
   ------- comparações executa Ω(n*lg n) comparações no pior caso.

==========
PROVA:

Nós provaremos o enunciado para o caso particular de instâncias sem elementos repetidos, caso em que cada operação de comparação entre elementos é equivalente a uma comparação da forma "x < y" (já que não faz sentido ter comparações do tipo "x = y" ou "x != y"). Naturalmente, uma vez provado que todo algoritmo baseado em comparações executa em tempo Ω(n*lg n) mesmo nesse caso restrito, segue imediatamente que nenhum algoritmo baseado em comparações pode executar em o(n*lg n) no caso geral, como desejado.

Considere a árvore de decisão de um tal algoritmo, para o caso da ordenação de "n" elementos. Suponha que a árvore em questão tem altura "h" e "f" folhas. Como o algoritmo é correto, então todas as n! permutações dos "n" elementos da entrada aparecem como folhas da árvore de decisão em questão, isto é, f >= n!. Além disso, como a árvore de decisão em questão é uma árvore estritamente binária (isto é, cada nó tem 0 ou 2 filhos), então ela tem no máximo 2^h folhas, isto é, f <= 2^h. Logo, n! <= 2^h ∴ lg n! <= h. Finalmente, pelo lema 1, temos lg n! = θ(n*lg n), e portanto h = Ω(n*lg n). Assim, como a altura da árvore de decisão do algoritmo é o número de comparações realizadas pelo algoritmo no pior caso, então esse número é Ω(n*lg n), CQD.
==========

 ORDENANDO EM MENOS QUE n*lg n

É importante observar que o limite inferior acima somente se aplica a algoritmos baseados exclusivamente em comparações. O algoritmo abaixo, por exemplo, resolve o problema mais restrito de ORDENAR UM CONJUNTO DE INTEIROS DE 1 A "k" sem realizar qualquer comparação direta entre os elementos do vetor de entrada:

```
================================================================================
Algoritmo:    counting_sort_1
Entrada:      um vetor V[1..n] de inteiros e um natural positivo "k".
Pré-condição: ∀ 1 <= i <= n, 1 <= V[i] <= k.
Saída:        nenhuma.
Pós-condição: V[1..n] estará ordenado.
--------------------------------------------------------------------------------
01. PARA i DE 1 A k
02. | aux[i] := 0
03. PARA i DE 1 A n
04. | ++aux[V[i]]
    // Há aux[j] chaves (iguais a) "j" em "V".
05. qtd_ant := aux[1] , aux[1] := 1 , pos_correta[1] := aux[1]
06. PARA i DE 2 A k
07. | qtd_atual := aux[i]
08. | aux[i] := aux[i-1] + qtd_ant
09. | qtd_ant := qtd_atual
10. | pos_correta[i] := aux[i]
    // No vetor ordenado, as chaves "j" (se houver) começam na posição aux[j].
11. x := 1  // Armazena a chave correta para a posição "i" do vetor ordenado.
12. PARA i DE 1 A n
13. | ENQUANTO x < k E aux[x+1] <= i
14. | | ++x
15. | ENQUANTO V[i] != x
16. | | p := pos_correta[V[i]] , ++pos_correta[V[i]]
17. | | temp := V[p] , V[p] := V[i] , V[i] := temp
================================================================================
```

É fácil verificar que o algoritmo acima utiliza θ(k) de memória auxiliar: são dois vetores [1..k] mais um número constante de variáveis inteiras. O algoritmo também executa em tempo θ(n+k), mas a análise não é igualmente direta, devido aos dois laços ENQUANTO internos ao último laço PARA do algoritmo. Basicamente, podemos argumentar da seguinte maneira:

1. Durante toda a execução do algoritmo, o laço ENQUANTO da linha 13 executa no máximo "k-1" iterações completas, pois o valor de "x" começa em 1, nunca diminui durante o algoritmo, aumenta em uma unidade a cada iteração do laço em questão e não ultrapassa o valor de "k".

2. O laço ENQUANTO da linha 15 executa no máximo "n" iterações durante todo o algoritmo, pois, em cada iteração, um elemento V[i] que não está numa posição correta é colocado numa posição correta. Como "V" possui apenas "n" elementos, então o laço em questão somente poderia executar mais que "n" iterações se um elemento, uma vez colocado numa posição correta, pudesse de lá ser retirado. Entretanto, isso somente poderia acontecer por meio do laço da linha 15, e nele não ocorre, pois nele apenas elementos V[i] em posição incorreta são movimentados para a frente no vetor, e elementos colocados para a frente nunca são movidos para trás, devido ao incremento "++pos_correta[V[i]]" da linha 16.

O algoritmo acima ordena um vetor sem realizar comparações diretas entre os elementos deste, por meio de uma contagem de quantos elementos há no vetor para cada valor de chave possível (dado que as chaves possíveis são facilmente determinadas pela entrada do algoritmo). A mesma ideia leva a um algoritmo mais simples se, ao invés de um segundo vetor auxiliar de tamanho "k", nós utilizarmos um segundo vetor auxiliar de tamanho "n", para guardar os elementos em suas posições corretas durante o segundo percurso de "V":

```
================================================================================
Algoritmo:    counting_sort_2
Entrada:      um vetor V[1..n] de inteiros e um natural positivo "k".
Pré-condição: ∀ 1 <= i <= n, 1 <= V[i] <= k.
Saída:        nenhuma.
Pós-condição: V[1..n] estará ordenado.
--------------------------------------------------------------------------------
01. PARA i DE 1 A k
02. | aux[i] := 0
03. PARA i DE 1 A n
04. | ++aux[V[i]]
    // Para todo 1 <= x <= k, "x" ocorre aux[x] vezes em V.
05. PARA i DE 2 A k
06. | aux[i] := aux[i] + aux[i-1]
    // Para todo 1 <= x <= k, há aux[x] elementos <= x em V. Em outras
    // palavras, no vetor ordenado, as chaves "x" terminam na posição aux[x].
07. PARA i DE n A 1
08. | W[aux[V[i]]] := V[i]
09. | --aux[V[i]]
10. PARA i DE n A 1
11. | V[i] := W[i]
================================================================================
```

É imediato observar que o algoritmo acima executa em tempo θ(n+k) e utiliza θ(n+k) de memória auxiliar. Em particular, se k <= n, então o algoritmo executa em tempo θ(n), o que é assintoticamente ótimo e melhor que o tempo de pior caso dos algoritmos baseados em comparação que havíamos considerado anteriormente. O mesmo vale para conjuntos de instâncias para as quais k = O(n), isto é, nas quais o maior elemento do vetor de entrada é assintoticamente limitado superiormente por "n". Observe que não é todo conjunto de instâncias que satisfaz a restrição k = O(n): considere, por exemplo, o conjunto dos vetores A_n[1..n] tais que, para todo n, o vetor A_n é tal que, para todo j < n, A_n[j] = j, e tal que A_n[n] = 2^n; claramente, não é possível limitar superiormente o maior elemento de cada vetor A_n por meio de uma função linear em "n".

3. DEFINIÇÃO: um algoritmo de ordenação é estável sse, em toda execução dele,
   --------- se X e Y são elementos do vetor de entrada que possuem a mesma chave e se X aparece antes de Y nesse vetor, então X também aparece antes de Y no vetor ordenado.

4. EXERCÍCIO: os algoritmos counting_sort_1 e counting_sort_2 são estáveis?
   --------- E quanto a heapsort, mergesort e quicksort?

5. EXERCÍCIO (IMPORTANTE): escreva uma versão generalizada de counting_sort_2,
   ---------------------- que receba como entrada um vetor de números inteiros no intervalo [LI..LS], sendo os inteiros LI e LS também parte da entrada.

6. EXERCÍCIO (IMPORTANTE): o laço da linha 7 de counting_sort_2 poderia ser
   ---------------------- escrito "PARA i DE 1 A n" sem prejuízo algum para as propriedades desejadas para o algoritmo?

7. EXERCÍCIO: escreva uma variação de "counting_sort_2" que é estável e que
   --------- preenche o vetor W do início para o fim, ao invés do contrário, como é feito no algoritmo original.

ORDENAÇÃO POR CAMPOS OU DÍGITOS

Uma maneira de ordenar valores compostos (como datas, palavras, etc; em geral, dados que obedeçam a uma ordem lexicográfica) é ordená-los várias vezes, cada vez com relação a um campo diferente, até que todos os campos tenham sido ordenados.

Talvez a maneira mais intuitiva de materializar a estratégia acima seja a seguinte. Considere, por exemplo, o problema de ordenar um conjunto de datas: pode-se fazer isso primeiramente ordenando todas as datas pelo ano; em seguida, para cada grupo de datas com o mesmo valor de ano, deve-se ordenar o grupo com relação ao mês; por fim, para cada grupo de datas com o mesmo valor de ano e mês, deve-se ordenar o grupo com relação ao dia.

Observe que a estratégia acima também pode ser utilizada para ordenar números inteiros: basta considerar cada inteiro como uma sucessão de dígitos em alguma base maior que um, e fazer a ordenação dígito a dígito.

8. EXERCÍCIO: considere a aplicação da estratégia acima para a ordenação de
   --------- chaves que consistem em "d" dígitos, cada um assumindo um valor de 1 a "k". Nesse caso, prove que, para instâncias em que, para cada um dos "d" dígitos, toda chave participa de algum grupo que será ordenado com relação ao dígito em questão, então o algoritmo executa em tempo $\Omega(d(n+k))$, supondo que a ordenação de cada grupo acontece por contagem ("counting sort").

Outra maneira de materializar a estratégia apresentada no início desta seção é ordenar os campos do menos significativo ao mais significativo, incluindo TODAS AS CHAVES em cada ordenação; se cada ordenação for estável (ver definição 3 acima), então o conjunto das chaves estará corretamente ordenado ao final. Assim, por exemplo, para ordenar um conjunto de datas, basta primeiro ordená-las todas por dia, depois todas por mês e finalmente todas por ano. No caso em que as chaves são dígitos, o algoritmo é simplesmente o seguinte:

```
==========================================================================
Algoritmo:    radix_sort
Entrada:      vetor V[1..n], a chave de cada elemento consistindo em "d"
              dígitos, cada dígito assumindo valores de 1 a "k" (ou 0 a k-1).
Pré-condição: nenhuma.
Saída:        nenhuma.
Pós-condição: o vetor estará ordenado.
--------------------------------------------------------------------------
1. PARA cada dígito "i", do menos significativo ao mais significativo:
2. | ordene V com relação ao dígito "i" usando ordenação por contagem.
==========================================================================
```

Observe que, acima, o algoritmo usado para a ordenação de cada dígito é (a ordenação por contagem, que é) estável, o que é necessário para que o algoritmo "radix_sort" seja correto. O uso da ordenação por contagem também implica que o tempo de ordenar cada dígito é $\theta(n+k)$, e portanto que o algoritmo radix_sort executa em tempo $\theta(d(n+k))$.

9. EXERCÍCIO: qual é o tempo de execução de radix_sort para ordenar "n" naturais
   --------- armazenados em 32 bits? Para que valores de "n" você usaria a ordenação por dígitos ao invés de um algoritmo de ordenação $\theta(n*\lg n)$?