```
1. TEOREMA: todo algoritmo de ordenação correto e baseado em
   comparações executa Ω(n*lg n) comparações.

=================================================================
----------------------------- PROVA -----------------------------
Considere a árvore de decisão de um tal algoritmo,
  para o caso da ordenação de n elementos.
(Como discutido em sala, nós supomos que os números da entrada
  são todos distintos e que a única operação de comparação
  utilizada é a avaliação de uma expressão da forma "x < y".)
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, então ela tem no máximo 2^h folhas,
  isto é, f <= 2^h.
Logo, n! <= 2^h ∴ lg n! <= h.
Finalmente, como lg n! = θ(n * lg n) (veja abaixo uma demonstração),
  então h = Ω(n * lg n), ou seja, o tempo de execução do algoritmo,
  no pior caso, é Ω(n*lg n), CQD.
----------------------------- PROVA -----------------------------
=================================================================


2. 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(n!)  = lg(1) + lg(2) + ... + lg(n)
=> lg(n!) <= 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
  constantes c > 0 (real) e n_0 > 0 (natural) 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 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:

$$
\begin{aligned}
& \lg(n) \le \lg(i) + \lg(n-i+1) \\
\iff & \lg(n) \le \lg(i*(n-i+1)) \\
\iff & n \le i*(n -i +1) \\
\iff & n \le in -i^2 +i \\
\iff & i^2 -i \le in -n \\
\iff & i(i-1) \le n(i-1) \\
\iff & i \le n, \text{ o que é verdade, CQD.}
\end{aligned}
$$

----------------------------- PROVA -----------------------------
===

3. Algoritmo de ordenação por contagem:

```
============================================================
Algoritmo: ord_cont
Entrada: vetor A[1..n] de números naturais de 1 a k,
         vetor auxiliar O[1..k] de naturais,
         vetor auxiliar B[1..n] de naturais.
Saída: nenhuma.
------------------------------------------------------------
1. PARA i de 1 a k
2. | O[i] := 0
3. PARA i de 1 a n
4. | ++O[A[i]]

   // FATO: para todo x <= k, x ocorre O[x] vezes em A.

5. PARA i de 2 a k
6. | O[i] := O[i] + O[i-1]

   // FATO: para todo x <= k, há O[x] elementos <= x em A.

7. PARA i de n a 1
8. | B[O[A[i]]] := A[i]
9. | --O[A[i]]
============================================================
```

4. É imediato observar que o algoritmo acima executa em tempo $\Theta(n+k)$.
   Em particular, se k <= n, então o algoritmo executa em tempo $\Theta(n)$, o que é assintoticamente melhor que o tempo levado pelos algoritmos baseados em comparação que havíamos considerado (merge sort, etc).
O mesmo vale para conjuntos de instâncias para os quais k = O(n), isto é, para os quais o maior elemento da entrada esteja 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".

5. EXERCÍCIO: escreva uma versão generalizada do algoritmo acima, que receba como entrada um vetor de números inteiros quaisquer (e que, portanto, podem ser iguais a zero ou mesmo negativos).

6. EXERCÍCIO: experimente obter invariantes para o algoritmo acima, de forma a argumentar precisamente que ele é correto.

7. EXERCÍCIO: uma maneira de ordenar valores compostos (como datas, palavras, etc) é ordená-los várias vezes, cada uma com relação a uma chave diferente.

Assim, por exemplo, para ordenar um conjunto de datas, basta ordená-las primeiro por dia, depois por mês e finalmente por ano (também é possível fazer na ordem contrária, mas seria mais complicado gerenciar o processo, não?).

Assim, suponha que cada elemento de um vetor A[1..n] possui uma chave composta de "d" campos, A[i].chave[1] ... A[i].chave[d], que possuem o mesmo tipo e podem assumir valores de 1 a "k".

Escreva então um algoritmo que ordena o vetor A em tempo O(d(n+k)), considerando que chave[1] é a chave mais significativa.

8. EXERCÍCIO: utilize o algoritmo acima para produzir um algoritmo que ordena inteiros levando em consideração que cada inteiro está armazenado em "b" bits, e que, para cada r de 1 a b, um inteiro pode ser visto como uma chave composta de aproximadamente b/r campos, cada um assumindo 2^r valores possíveis (cada campo corresponde a um trecho de "r" bits consecutivos).

Qual é o tempo de execução do seu algoritmo?