

Algoritmos de Ordenação

Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

24 de agosto de 2023

Introdução

Leitura(s) Recomendada(s)



Seções 3.1.2, 11.1 (*Merge Sort*), 11.2 (*Quick Sort*), 11.3.3 (comparação)

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados e algoritmos em Java**.

Tradução: Bernardo Copstein. 5. ed. Porto Alegre: Bookman, 2013. xxii, 713 p. E-book. ISBN

9788582600191. Tradução de: Data Structures and Algorithms in Java, 5th Edition. Disponível em:

<<https://integrada.minhabiblioteca.com.br/#/books/9788582600191/>>. Acesso em: 01 ago. 2023.

Sites sobre Ordenação [*]

- Animações:
<http://www.sorting-algorithms.com/>
- Algoritmos na wikipedia:
https://en.wikipedia.org/wiki/Sorting_algorithm
- Danças:
<http://makezine.com/2011/04/12/data-sorting-dances/>
- 15 algoritmos em 6 minutos:
<https://www.youtube.com/watch?v=kPRAOW1kECg>
- Visualização e comparação de algoritmos de ordenação:
<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>
- Visualização *Bubble Sort vs Quick Sort*:
<https://www.youtube.com/watch?v=aXXWXz5rF64>
- Visualização *Merge Sort vs Quick Sort*:
<https://www.youtube.com/watch?v=es2T6KY45cA>

Revisão: Algoritmos de Pesquisa

- Pesquisa Linear

- Pode ser aplicada sobre qualquer coleção, ordenada ou não
- Procura um item, comparando-o com cada elemento da coleção, até achar ou chegar no final
- Melhor caso: o item procurado está na primeira posição da coleção
- Pior caso: o item NÃO está na coleção
- Complexidade: $O(n)$

- Pesquisa Binária

- A coleção deve estar ordenada
- Estratégia básica:
 - Verifica o elemento central: se encontrou, a busca termina
 - Se o item for menor que o central, considera apenas a parte abaixo do elemento central
 - Se o item for maior que o central, considera apenas a parte acima do elemento central
- Trabalha subdividindo a coleção e reaplicando sempre a estratégia básica, o que o torna adequado para implementação recursiva
- Complexidade: $O(\log n)$

Algoritmos de Ordenação

Algoritmos de Ordenação

- Organizam os elementos de uma coleção segundo determinado critério (ordem crescente de valor, por exemplo)
- Operação básica: troca de elementos
- Exemplos
 - *Bubble Sort*
 - *Selection Sort*
 - *Insertion Sort*
 - *Merge Sort*
 - *Quick Sort*
 - etc.
- Em geral, os mais simples nem sempre tem bom desempenho (menos otimizados)
- Algoritmos com bom desempenho costumam ser mais sofisticados
- São importantes quando se quer implementar busca eficiente (pesquisa binária)

Bubble Sort

Bubble Sort

- É um dos métodos mais simples de ordenação
- Estratégia: compara elementos adjacentes, e, se estiverem fora de ordem, troca os elementos
- Repete-se a estratégia básica até que a coleção esteja ordenada
- Complexidade: $O(n)$ (melhor caso) ou $O(n^2)$ (pior caso)

Bubble Sort: Exemplo

0	1	2	3	4	5	6	7	8	9
5	7	8	1	10	9	4	6	3	2
5	7	1	8	9	4	6	3	2	10
5	1	7	8	4	6	3	2	9	10
1	5	7	4	6	3	2	8	9	10
1	5	4	6	3	2	7	8	9	10
1	4	5	3	2	6	7	8	9	10
1	4	3	2	5	6	7	8	9	10
1	3	2	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Bubble Sort: Implementação

```
void bubbleSort(int *dados, int tam) {  
    int trocou;  
    do {  
        trocou = 0;  
        --tam;  
        for (int i=0; i<tam; ++i) {  
            if (dados[i] > dados[i+1]) {  
                int aux = dados[i];  
                dados[i] = dados[i+1];  
                dados[i+1] = aux;  
                trocou = 1;  
            }  
        }  
    } while (trocou);  
}
```

Bubble Sort: Mais informações [*]

- <http://www.sorting-algorithms.com/bubble-sort>
- <https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/>

Selection Sort

Selection Sort

- É um algoritmo de ordenação por seleção
- Fácil de implementar e bastante intuitivo, o que não garante eficiência...
- Estratégia: procurar o menor elemento e colocá-lo na sua posição
- Repete-se a estratégia até que todos os elementos estejam em sua posição
- Complexidade: $O(n^2)$ (melhor e pior caso)

Selection Sort: Exemplo

0	1	2	3	4	5	6	7	8	9
5	7	8	1	10	9	4	6	3	2
1	7	8	5	10	9	4	6	3	2
1	2	8	5	10	9	4	6	3	7
1	2	3	5	10	9	4	6	8	7
1	2	3	4	10	9	5	6	8	7
1	2	3	4	5	9	10	6	8	7
1	2	3	4	5	6	10	9	8	7
1	2	3	4	5	6	7	9	8	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Selection Sort: Implementação

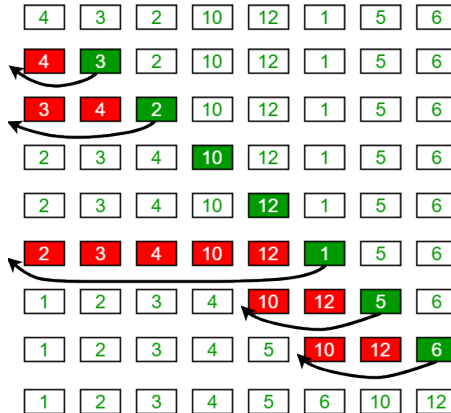
```
void selectionSort(int *dados, int tam) {  
    for (int i=0; i<tam-1; ++i) {  
        int men = i;  
        for (int j=i+1; j<tam; ++j)  
            if ( dados[j] < dados[men] ) men = j;  
        if ( men != i ) {  
            int aux = dados[men];  
            dados[men] = dados[i];  
            dados[i] = aux;  
        }  
    }  
}
```


Insertion Sort

Insertion Sort

- É um algoritmo de ordenação por inserção
- Estratégia:
 - Escolhe-se uma base que inicia no segundo elemento e avança até o último elemento
 - Sempre à esquerda da base todos os elementos devem estar ordenados
 - Busca-se a posição da base nos elementos à esquerda, sempre deslocando os elementos uma posição para a direita enquanto não chegar na posição correta da base
 - Quando chegar na posição correta da base, atribui-se o valor da base para esta posição
- Trata-se de um algoritmo um pouco mais avançado do que os dois anteriores
- Complexidade: $O(n)$ (melhor caso) ou $O(n^2)$ (pior caso)

Insertion Sort: Exemplo 1



Fonte: <https://www.geeksforgeeks.org/insertion-sort/>

Insertion Sort: Exemplo 2

0	1	2	3	4	5	6	7	8	9
5	7	8	1	10	9	4	6	3	2
5	7	8	1	10	9	4	6	3	2
5	7	8	1	10	9	4	6	3	2
1	5	7	8	10	9	4	6	3	2
1	5	7	8	10	9	4	6	3	2
1	5	7	8	9	10	4	6	3	2
1	4	5	7	8	9	10	6	3	2
1	4	5	6	7	8	9	10	3	2
1	3	4	5	6	7	8	9	10	2
1	2	3	4	5	6	7	8	9	10

Insertion Sort: Implementação

```
void insertionSort(int *dados, int tam) {  
    for (int i=1; i<tam; ++i) {  
        int base = dados[i];  
        int j = i-1;  
        while ( j>=0 && base < dados[j] ) {  
            dados[j+1] = dados[j];  
            --j;  
        }  
        dados[j+1] = base;  
    }  
}
```

Insertion Sort: Mais informações [*]

- <http://www.sorting-algorithms.com/insertion-sort>
- <https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/tutorial/>

Merge Sort

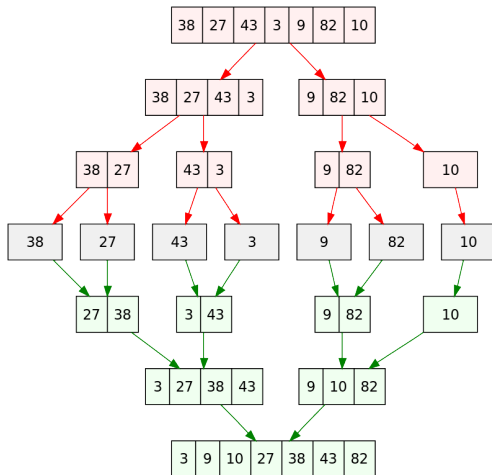
Merge Sort

- É um algoritmo de ordenação por intercalação
- Utiliza o padrão (estratégia) conhecido como “divisão e conquista”
- Consiste de 3 etapas
 - Divisão: se há algo a ordenar, divide os dados de entrada em duas (ou mais) partes e executa o algoritmo sobre cada uma das partes; se não há nada a ordenar, retorna a solução
 - Conquista: cada parte dos dados é classificada recursivamente
 - Combinação: quando cada subconjunto está classificado (internamente), eles devem ser combinados (*merge*) realizando-se uma intercalação
- Permite implementação recursiva

Merge Sort: Estratégia [*]

- Para ordenar uma sequência S com n elementos:
 - **Dividir**: se S tem zero ou um elemento, retorna S , pois já está classificado; senão, remove os elementos de S e coloca-os em duas sequências, S_1 e S_2 ($n/2$ elementos em cada um)
 - **Conquistar**: classifica as sequências S_1 e S_2 recursivamente
 - **Combinar**: coloca os elementos de volta em S com a união das sequências S_1 e S_2 ordenadas

Merge Sort: Exemplo



Fonte: https://en.wikipedia.org/wiki/Merge_sort

- A execução do algoritmo pode ser vista como uma árvore binária
- Cada nodo representa uma chamada recursiva do algoritmo *Merge Sort*
- Nodos recebem sequências de entrada para serem processadas e, por fim, geram sequências de saída ordenadas

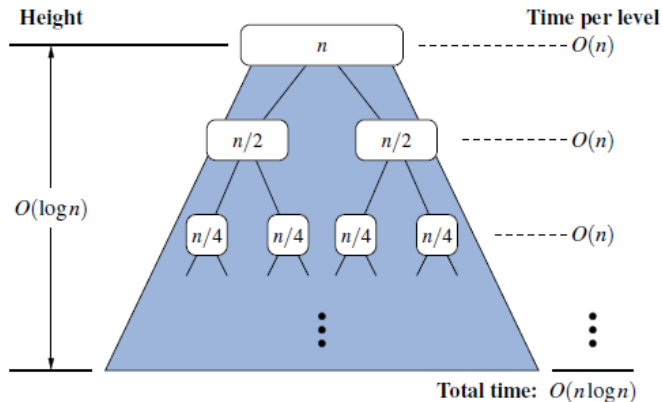
Merge Sort: Implementação

```
void merge(int *dados, int ini, int meio, int fim) {
    int p = ini, q = meio+1, k=0;
    int *aux = new int[fim-ini+1];
    for (int i = ini; i <= fim; i++){
        if (p > meio)                aux[k++] = dados[q++];
        else if (q > fim)            aux[k++] = dados[p++];
        else if( dados[p] < dados[q]) aux[k++] = dados[p++];
        else                        aux[k++] = dados[q++];
    }
    for (int p=0; p<k; p++) dados[ini++] = aux[p];
    delete[] aux;
}

void mergeSort(int *dados, int ini, int fim) {
    if ( ini >= fim ) return;
    int meio = (ini + fim) / 2;
    mergeSort(dados, ini, meio);
    mergeSort(dados, meio+1, fim);
    merge(dados,ini,meio,fim);
}
```

Merge Sort: Desempenho [*]

- O tamanho da sequência de entrada é a metade a cada chamada recursiva
- A árvore associada a uma execução do algoritmo com uma sequência de tamanho n , tem altura $\log n$
- Conclusões:
 - Altura da árvore é $\log n$
 - Tempo gasto em cada nível: $O(n)$
 - Tempo de execução: $O(n \log n)$



Merge Sort: Mais informações [*]

- <http://www.sorting-algorithms.com/merge-sort>
- <https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/tutorial/>

Quick Sort

Quick Sort

- Foi criado pelo cientista britânico Charles Antony Richard Hoare em 1959 e publicado em 1961
- É um algoritmo de ordenação por comparação que também utiliza o padrão “Divisão e Conquista”
- Diferentemente do *Merge Sort*, o *Quick Sort* procura realizar a parte mais significativa do processamento **antes** das chamadas recursivas
- A estratégia geral subdivide-se nos seguintes passos:
 - Escolhe-se um elemento da lista, denominado **pivô**
 - Particionamento: a lista é reorganizada de forma que todos os elementos da lista menores que o pivô fiquem à esquerda do pivô e que todos os elementos maiores do que o pivô fiquem à sua direita – o pivô estará em sua posição correta, porém “cercado” por duas listas não ordenadas
 - Essas duas listas são recursivamente ordenadas usando a mesma estratégia
- Há variantes relacionadas principalmente à escolha do pivô, que o influencia o desempenho

Quick Sort: Escolha do Pivô

- A escolha do pivô é influencia o desempenho
- Métodos:
 - Hoare: escolhe o elemento central da coleção como pivô
 - Lomuto: criado por Nico Lomuto, tipicamente escolhe o primeiro ou último elemento da coleção
 - Pivô aleatório
 - Valor intermediário entre primeiro, central e último elementos

Quick Sort: Algoritmo (Hoare) [Fonte: <https://pt.wikipedia.org/wiki/Quicksort>]

```

procedimento QuickSort(X[], IniVet, FimVet)
var
    i, j, pivo, aux
início
    i <- IniVet
    j <- FimVet
    pivo <- X[(IniVet + FimVet) div 2]
    enquanto(i <= j)
    |
        enquanto (X[i] < pivo) faça
        |     i <- i + 1
        |
        fimEnquanto
        enquanto (X[j] > pivo) faça
        |     j <- j - 1
        |
        fimEnquanto
        se (i <= j) então
        |     aux <- X[i]
        |     X[i] <- X[j]
        |     X[j] <- aux
        |     i <- i + 1
        |     j <- j - 1
        |
        fimSe
    fimEnquanto
    se (IniVet < j) então
    |     QuickSort(X, IniVet, j)
    fimSe
    se (i < FimVet) então
    |     QuickSort(X, i, FimVet)
    fimSe
fimProcedimento

```

Quick Sort: Algoritmo (Lomuto) [Fonte: <https://pt.wikipedia.org/wiki/Quicksort>]

```
algorithm quicksort(A, lo, hi) is
  if lo < hi then
    p := particiona(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)
```

```
algorithm particiona(A, lo, hi) is
  pivot := A[hi]
  i := lo - 1
  for j := lo to hi - 1 do
    if A[j] < pivot then
      i := i + 1
      swap A[i] with A[j]
  if pivot < A[i + 1] then
    swap A[i + 1] with A[hi]
  return i + 1
```

Quick Sort: Análise

- Se a **posição do pivô** escolhido for **central**, o *Quick Sort* consegue duas subcoleções de tamanhos próximos
 - Se isso **sempre** ocorresse, a altura da árvore de execução seria $O(\log n)$
 - Como em cada nível o tempo de execução será $O(n)$
 - Então, o **melhor** tempo de execução do algoritmo poderia ser definido como $O(n \log n)$
- Por outro lado, no caso da variante Lomuto, que escolhe o último elemento como pivô, se a coleção já estiver ordenada, tem-se uma situação de pior caso, ou seja, $O(n^2)$
- Portanto, o desempenho do *Quick Sort* está fortemente atrelado à escolha do pivô e à configuração ou posição inicial dos elementos na coleção

Quick Sort: Mais informações [*]

- <https://en.wikipedia.org/wiki/Quicksort>
- <http://www.sorting-algorithms.com/quick-sort>
- <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>

Comparação

Comparação

- Definir qual é o “melhor” algoritmo de ordenação nem sempre é fácil
- Há muitas variações, tanto nas implementações dos algoritmos quanto nas configurações de coleções (ordendada, invertida, aleatória, com muitos elementos duplicados, sem elementos duplicados, etc.)
- Até mesmo o *Bubble Sort* pode apresentar o melhor desempenho para uma configuração específica
- Mas há alguns fatores que devem ser considerados...

Estabilidade [*]

- Um algoritmo de ordenação é estável (*stable*) se não altera a posição relativa dos elementos que têm o mesmo valor
- Exemplo:
 - Coleção inicial:
`{ {"João", 21}, {"Ana", 55}, {"João", 13"}, {"Beto", 34}, {"Yuri", 23} }`
 - Coleção ordenada por um algoritmo estável:
`{ {"Ana", 55}, {"Beto", 34}, {"João", 21}, {"João", 13"}, {"Yuri", 23} }`
 - Coleção ordenada por um algoritmo instável:
`{ {"Ana", 55}, {"Beto", 34}, {"João", 13"}, {"João", 21}, {"Yuri", 23} }`

Comparação dos Algoritmos quanto à Estabilidade

- São estáveis:
 - *Bubble Sort*
 - *Insertion Sort*
 - *Merge Sort*
- São instáveis:
 - *Selection Sort*
 - *Quick Sort*

Armazenamento

- Versões recursivas dos algoritmos necessitarão de memória da pilha
 - No *Merge Sort* e *Quick Sort*, o número máximo de chamadas recursivas aninhadas (ativas em determinado momento), em geral, será $O(\log n)$
 - Cuidado: implementações recursivas de outros algoritmos de ordenação podem gerar $O(n)$ chamadas recursivas aninhadas
- Algoritmos de ordenação *in-place* utilizam apenas o espaço da própria coleção para realizar a ordenação, NÃO necessitando de áreas de memória auxiliares
 - *Merge Sort*, por exemplo, precisa de um vetor auxiliar

Bubble Sort

- Complexidade: $O(n^2)$ (pior caso) ou $O(n)$ (melhor caso, considerando a implementação otimizada)
- É simples de ser implementado
- É estável
- Não necessita de um vetor auxiliar (*in-place*), ocupando menos memória
- NÃO é recomendado para coleções grandes

Selection Sort

- Complexidade: $O(n^2)$
- É simples de ser implementado
- Não necessita de um vetor auxiliar (*in-place*), ocupando menos memória
- É relativamente rápido para pequenas coleções
- É um dos mais lentos para coleções grandes
- NÃO é estável
- Executa SEMPRE $(n^2 - n)/2$ comparações

Insertion Sort

- Complexidade: $O(n^2)$ (pior caso) ou $O(n)$ (melhor caso)
- É estável
- Não necessita de um vetor auxiliar (*in-place*), ocupando menos memória
- Tem desempenho razoável principalmente para pequenas coleções (tamanho da ordem de dezenas)
- Eficiente para ordenação de coleções “quase” ordenadas

Merge Sort

- Complexidade: $O(n \log n)$ (pior caso)
- É estável
- Tem bom desempenho para coleções grandes
- Necessita de um vetor auxiliar

Quick Sort

- Complexidade: $O(n \log n)$ (melhor caso) ou (n^2) (pior caso)
- Tem um desempenho excelente para vetores grandes
- NÃO é estável
- A escolha do pivô pode comprometer o desempenho
- Análises experimentais mostram que, se a sequência de entrada couber na memória principal, versões *in-place* do *Quick Sort* executam mais rápido que o *Merge Sort* [*]

Comparativo com Medição de Tempo

- Vetor de 10.000 inteiros em 3 configurações: ordenado, invertido e aleatório
- Tempos medidos em μs (microssegundos)

Algoritmo	Ordenado	Invertido	Aleatório
<i>Bubble Sort</i>	117	192173	224400
<i>Selection Sort</i>	135131	105908	106956
<i>Insertion Sort</i>	174	148203	58291
<i>Merge Sort</i>	4277	3879	6470
<i>Quick Sort</i> (Hoare)	1186	406	1587
<i>Quick Sort</i> (Lomuto)	203035	136896	1108

Exercícios e Testes

Exercício 1

- 1 Implemente os algoritmos das duas versões de *Quick Sort* apresentadas nestas lâminas em C/C++ e teste as suas implementações com o código da página a seguir.

Exercício 1

```

#include <iostream>
#include <cstdlib>
#include <sys/time.h>
#define TAM 10000
using namespace std;
void quickSort(int *dados, int ini, int fim);

int estaOrdenado(int *dados, int tam) {
    for (int i=0; i<tam-1; ++i) if (dados[i] > dados[i+1]) return 0;
    return 1;
}

int main() {
    int vetor[TAM];
    struct timeval antes, depois;
    unsigned long microssegundos;
    srand(time(0));

    cout << ">_Ordenado:_";
    for (int i=0; i<TAM; i++) vetor[i] = i; //preenche o vetor ordenado
    gettimeofday(&antes, NULL); quickSort(vetor,0,TAM-1); gettimeofday(&depois, NULL);
    microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 + depois.tv_usec - antes.tv_usec;
    if (estaOrdenado(vetor,TAM)) cout << microssegundos << "_us" << endl; else cout << "FALHOU" << endl;

    cout << ">_Invertido:_";
    for (int i=0; i<TAM; i++) vetor[i] = TAM - i; //preenche o vetor invertido
    gettimeofday(&antes, NULL); quickSort(vetor,0,TAM-1); gettimeofday(&depois, NULL);
    microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 + depois.tv_usec - antes.tv_usec;
    if (estaOrdenado(vetor,TAM)) cout << microssegundos << "_us" << endl; else cout << "FALHOU" << endl;

    cout << ">_Aleatório:_";
    for (int i=0; i<TAM; i++) vetor[i] = rand()%TAM; //preenche o vetor aleatoriamente
    gettimeofday(&antes, NULL); quickSort(vetor,0,TAM-1); gettimeofday(&depois, NULL);
    microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 + depois.tv_usec - antes.tv_usec;
    if (estaOrdenado(vetor,TAM)) cout << microssegundos << "_us" << endl; else cout << "FALHOU" << endl;

    return 0;
}

```

Exercícios 2, 3 e 4

- 2 Adapte o código com a função `main()`, da lâmina anterior, para rodá-lo com as implementações de *Bubble Sort*, *Selection Sort*, *Insertion Sort* e *Merge Sort* (apresentadas nestas lâminas). Compare os tempos de execução de todas as implementações de algoritmos de ordenação.
- 3 O código da função `main()`, da lâmina anterior, executa testes com um vetor já ordenado (em ordem crescente), com um vetor invertido (ordenado em ordem decrescente) e com um vetor com valores aleatórios. Crie uma nova configuração de teste para um vetor com alto número de elementos duplicados, e teste todos os algoritmos para esta nova configuração.
- 4 Procure outros algoritmos de ordenação na Internet, implemente-os e adapte essas implementações para funcionarem no mesmo padrão dos testes executados anteriormente.

Créditos

Créditos

- Estas lâminas contêm trechos adaptados de materiais criados e disponibilizados pela professora Isabel Harb Manssour [*].

Soluções

Solução: *Quick Sort* (Hoare) em C/C++

```
void quickSort(int *dados, int ini, int fim) {
    int i = ini, j = fim, pivo = dados[(ini+fim)/2];
    while (i <= j) {
        while (dados[i] < pivo)
            ++i;
        while (dados[j] > pivo)
            --j;
        if (i <= j) {
            int aux = dados[i];
            dados[i] = dados[j];
            dados[j] = aux;
            ++i;
            --j;
        }
    }
    if (ini < j) quickSort(dados, ini, j);
    if (i < fim) quickSort(dados, i, fim);
}
```

Solução: *Quick Sort* (Lomuto) em C/C++

```
int particiona(int *dados, int ini, int fim) {
    int pivo = dados[fim];
    int i = ini-1;
    for (int j=ini; j<fim; ++j) {
        if (dados[j] < pivo) {
            ++i;
            int aux = dados[i]; dados[i] = dados[j]; dados[j] = aux;
        }
    }
    if (pivo < dados[i+1]) {
        int aux = dados[i+1]; dados[i+1] = dados[fim]; dados[fim] = aux;
    }
    return i+1;
}

void quickSort(int *dados, int ini, int fim) {
    if (ini < fim) {
        int pivo = particiona(dados, ini, fim);
        quickSort(dados, ini, pivo-1);
        quickSort(dados, pivo+1, fim);
    }
}
```