Algoritmos de Ordenação

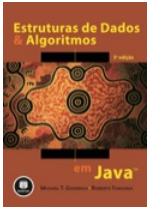
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

22 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=kPRA0W1kECg`
- 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 algorimo 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 posiçao
- 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

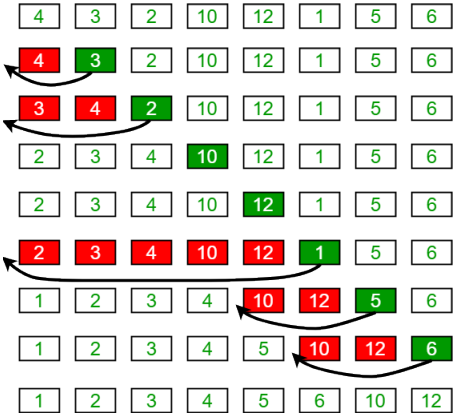
```
void selectionSort(int *dados, int tam) {
  for (int i=0; i<tam-1; ++i) {
      int men = i;
      for (int j=i+1; j<tam; ++j)
          if ( dados[j] < dados[men] ) men = j;
      if ( men != i ) {
         int aux = dados[men];
         dados[men] = dados[i];
         dados[i] = aux;
      }
 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

Insertion Sort

Insertion Sort

- É um algorimo 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 uma 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

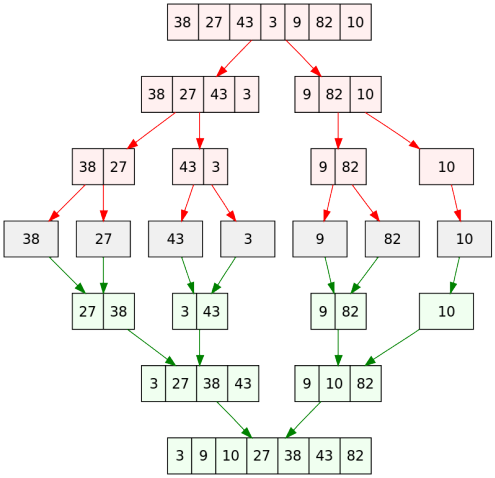
Merge Sort

- É um algorimo 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

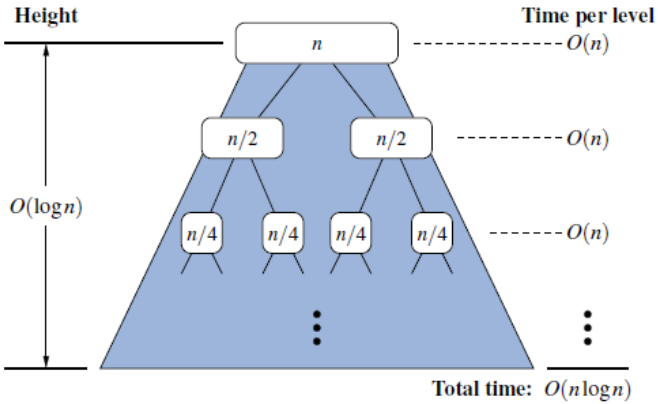
Merge Sort: Implementação

```
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 [*]

- Também utiliza o padrão "Dividir para Conquistar", mas de uma maneira diferente do *Merge Sort*
- O maior processamento é feito **antes** das chamadas recursivas
- Estratégia geral:
  - Divisão de S em subconjuntos (sequências)
  - Recursão para classificar cada subconjunto
  - Combinar as subsequências ordenadas através de uma concatenação simples

Quick Sort: 3 Etapas Principais [*]

1. Dividir
   - Se S tem pelo menos dois elementos, seleciona um deles para ser o **pivô**
   - O pivô pode ser qualquer elemento de S
   - Remove todos os elementos de S e coloca-os em três sequências:
     - L: armazena os elementos de S menores que o pivô
     - E: armazena os elementos de S iguais ao pivô
     - G: armazena os elementos de S maiores que o pivô
2. Conquistar
   - Recursivamente ordena as sequências L e G
3. Combinar
   - Coloca de volta os elementos em S, inserindo primeiro os elementos de L, depois de E e finalmente de G

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

Estabilidade [*]

- Um algoritmo de ordenação é estável (*stable*) se não altera a posição relativa dos elementos que têm o mesmo valor
- Exemplo:

  ```
  \item Coleção inicial:
  \begin{verbatim}
  { {"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*

Bubble Sort

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

Selection Sort

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

Insertion Sort

- Complexidade: $O(n^2)$ (pior caso) ou $O(n)$ (melhor caso)
- Vantagens:
  - É estável
- Desvantagens:
  - ...

Merge Sort

- Complexidade: $O(n \log n)$
- Vantagens:
  - É estável
- Desvantagens:
  - ...

Quick Sort

- Complexidade: $O(n \log n)$
- Vantagens:
  - ...
- Desvantagens:
  - NÃO é estável

Créditos

Créditos

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