Algoritmos e Estruturas de Dados II – Relatório (Ordenação)

Caio Eduardo Ferreira de Miranda (GRR20232359)
Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba, Brasil
caio.miranda@ufpr.br

Disclaimer **– Antes de se iniciar a leitura deste relatório, deve-se saber que: em todos os testes, foi utilizado um vetor de 80000, 100000 e 120000 elementos, todos com números aleatórios; para os algoritmos de busca, assume-se que o vetor está ordenado; todos os algoritmos descritos neste relatório foram implementados em C, de maneira recursiva; todos os testes foram feitos no Fedora Linux 38.**

I. Busca sequencial ingênua

A *busca sequencial ingênua* é um algoritmo de busca feito com a intenção de verificar os elementos do vetor sequencialmente, até que se ache o valor desejado ou um elemento menor que esse valor. Nas saídas abaixo, foi buscado um elemento qualquer que existe dentro do vetor:

80000 elementos:

```
Tempo de exec. da busca sequencial: 0.000565s
Comparações feitas: 45821
8 está no vetor, pos: 34179
```

100000 elementos:

```
Tempo de exec. da busca sequencial: 0.000742s
Comparações feitas: 57190
8 está no vetor, pos: 42810
```

120000 elementos:

```
Tempo de exec. da busca sequencial: 0.000891s
Comparações feitas: 68562
8 está no vetor, pos: 51438
```

Como se pode ver, o tempo de execução e o número de comparações crescem de acordo com o tamanho da entrada, de maneira controlada. Esse comportamento controlado se deve ao fato da *busca sequencial ingênua* ser um algoritmo linear, que "acompanha" o valor do tamanho da entrada.

II. Busca binária

A *busca binária* é um algoritmo de busca que divide o vetor pela metade repetidamente até que se ache o valor desejado (divisão e conquista).

Nas saídas abaixo, foi buscado um elemento qualquer que existe dentro do vetor:

80000 elementos:

```
Tempo de exec. da busca binária: 0.000001s
Comparações feitas: 16
8 está no vetor, pos: 34179
```

100000 elementos:

```
Tempo de exec. da busca binária: 0.000000s
Comparações feitas: 16
8 está no vetor, pos: 42810
```

120000 elementos:

```
Tempo de exec. da busca binária: 0.000001s
Comparações feitas: 17
8 está no vetor, pos: 51438
```

O tempo de execução foi quase imediato e o número de comparações foi baixíssimo, quase igual em todas as entradas. Isso ocorre pois o vetor sempre é dividido pela metade em cada chamada recursiva, fazendo apenas uma comparação por vez. Tem-se que o número de comparações para o algoritmo implementado sempre será algo no seguinte intervalo:

$$\log_2(n) \leq C(n) \leq 1 + \log_2(n)$$

Um algoritmo logarítmico, extremamente eficiente se comparado com a *busca sequencial ingênua*, por exemplo.

III. Insertion sort

O *insertion sort* é um algoritmo de ordenação que "insere" valores em sub-vetores ordenados que pertencem ao vetor principal, fazendo comparações até que o valor esteja inserido corretamente no sub-vetor da atual chamada recursiva. Esse processo ocorre recursivamente até que o vetor inteiro esteja ordenado. Abaixo, as saídas para a execução desse algoritmo:

80000 elementos:

```
Tempo de exec. do insertion sort: 7.582186s
Comparações feitas: 1526385908
```

100000 elementos:

```
Tempo de exec. do insertion sort: 11.822730s
Comparações feitas: 2373589323
```

120000 elementos:

```
Tempo de exec. do insertion sort: 16.966553s
Comparações feitas: 3413927126
```

Percebe-se que o número de comparações aumenta drasticamente com apenas algumas dezenas de milhares de elementos a mais, com o tempo de execução subindo em

mais que 3s por entrada. Isso ocorre pelo fato do *insertion sort* ser um algoritmo quadrático que realiza muitas trocas.

IV. Selection sort

O *selection sort* é um algoritmo de ordenação que seleciona o menor elemento do vetor e o coloca na primeira posição (considerando a atual chamada recursiva). O processo se repete até que o vetor esteja todo ordenado. Pode ser um algoritmo bem custoso (considerando o número de comparações), pois faz $n - 1$ comparações em todas as chamadas recursivas, sem exceção. Abaixo, as saídas para a execução do algoritmo:

80000 elementos:

```
Tempo de exec. do selection sort: 7.093860s
Comparações feitas: 3199960000
```

100000 elementos:

```
Tempo de exec. do selection sort: 11.037751s
Comparações feitas: 4999950000
```

120000 elementos:

```
Tempo de exec. do selection sort: 15.947230s
Comparações feitas: 7199940000
```

O número de comparações é demasiadamente grande para todas as entradas, inclusive maior do que todos os outros algoritmos descritos aqui. Entretanto, nota-se um evento interessante: o *insertion sort* é executado mais lentamente que o *selection sort,* apesar de ter um número de comparações menor.

V. Merge sort

O *merge sort* é um algoritmo de ordenação naturalmente recursivo que divide o vetor pela metade repetidamente até que seus elementos estejam trivialmente ordenados (caso base). Após isso, chama uma função (*merge*) que junta duas partições ordenadas em uma só partição, também ordenada. Isso ocorre para todas as partições formadas ao longo do algoritmo, do caso base até todo o vetor. Finalmente, após todas as junções, se tem o vetor ordenado. Abaixo, estão saídas para a execução do algoritmo:

80000 elementos:

```
Tempo de exec. do merge sort: 0.010585s
Comparações feitas: 1186128
```

100000 elementos:

```
Tempo de exec. do merge sort: 0.013062s
Comparações feitas: 1513149
```

120000 elementos:

```
Tempo de exec. do merge sort: 0.015901s
Comparações feitas: 1846310
```

Percebe-se que o *merge sort* é o mais eficiente dos algoritmos testados, pois o número de comparações é o menor de todos, assim como o tempo de execução (isso para todas as entradas). Entretanto, existe um *trade-off* existente neste algoritmo, que é o fato dele precisar de um vetor auxiliar para ser executado, assim consumindo mais espaço que os demais algoritmos.

VI. Quicksort

O *quicksort* é um algoritmo de ordenação que seleciona um pivô e particiona o vetor de maneira que todos os elementos à esquerda do pivô sejam maiores que ele e todos à direita sejam menores. Esse processo ocorre recursivamente até que o vetor todo esteja ordenado. Abaixo, as saídas para a execução do algoritmo:

80000 elementos:

```
Tempo de exec. do quicksort: 0.757217s
Comparações feitas: 152508444
```

100000 elementos:

```
Tempo de exec. do quicksort: 1.179453s
Comparações feitas: 238275744
```

120000 elementos:

```
Tempo de exec. do quicksort: 1.695138s
Comparações feitas: 343136404
```

Dentre os algoritmos *in-place,* este algoritmo é o mais eficiente, tanto no número de comparações quanto no tempo de execução. Em um cenário no qual não se tem muito espaço, o *quicksort* é uma alternativa válida ao *merge sort.* O fato do algoritmo ser logarítmico, com $n * \log_2(n)$ comparações, o faz muito eficiente na maioria dos casos.