Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação
DCC004 - Algoritmos e Estrutura de Dados II

TP02 - Algoritmos de Ordenação

Turma TW
Aluno: Matheus Henrique Antunes Lima
Professor: Adriano Alonso Veloso

Junho
2018

Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação
DCC004 - Algoritmos e Estrutura de Dados II

TP02 - Algoritmos de Ordenação

Documentação do Trabalho Prático 02 da disciplina
DCC004 - Algoritmos e Estrutura de Dados II

Turma TW

Aluno: Matheus Henrique Antunes Lima

Professor: Adriano Alonso Veloso

Junho
2018

Conteúdo

1 Introdução

O trabalho de ordenar uma coleção de objetos pode parecer simples em teoria. Basta, a partir de um critério, organizar os itens dessa coleção. Podemos por exemplo organizar um conjunto de livros alfabeticamente de acordo com seu título, autor ou talvez pela editora. Devemos decidir também em ordenar de maneira crescente, decrescente ou de acordo com alguma outra maneira. Os critérios são vários e escolher a melhor maneira de realizar a ordenação pode ser desafiador.

No âmbito da programação não é diferente. Existem vários algoritmos de ordenação implementados onde cada um com sua particularidade e método utilizado se torna útil para uma certa situação.

Neste trabalho vamos analisar alguns destes algoritmos e, através de seus tempos de execução, ver qual se torna mais eficiente em uma determinada configuração de tamanho e estado inicial de ordenação dos dados.

2 Desenvolvimento

Foram implementados, na linguagem C, um módulo, *sort*, com os algoritmos de ordenação já conhecidos *Select Sort*, *Insertion Sort*, *Bubble Sort*, *Shell Sort*, *Quick Sort* e *Heap Sort*. Em um outro módulo também foi implementado a estrutura a ser ordenada, *Item*, e um arranjo destes itens *ItemArray*. Cada *Item* possuí uma chave *id* que será nosso parâmetro para fazer a ordenação. Estão incluídos também na TAD *Item* os métodos:

- *printArray*, que imprime as chaves dos itens no arranjo.
- *fillArray*, que preenche o aranjo de tamanho n com itens de acordo com um parâmetro p que define se os itens devem ser preenchidos já com ordenação crescente, decrescente ou de maneira aleatória.
- e *copyArray*, que recebe dois arranjos de tamanho n e copia os valores do primeiro para o segundo.

3 Implementação

No programa principal, *main*, foram instanciados dois *ItemArray*. Um será usado como referência para que todos os testes usem o mesmo arranjo. O outro será organizado por um algoritmo de ordenação e depois, a partir do método *copyArray*, voltará ao estado inicial para que seja organizado pelo próximo algoritmo. Foi instanciado também uma variável t para registrar o tempo de execução de cada algoritmo usando o método *clock* da biblioteca *time.h*.
É passado por parâmetro, na chamada do programa principal, um n para o tamanho dos arranjos e um p para definir o estado inicial da ordenação dos arranjos, podendo ser ordenados crescente, decrescente ou aleatoriamente.

4 Análise dos Resultados

O programa principal foi executado um total de 20 vezes para os tamanhos 100, 1000, 10000 e 100000 itens no arranjo e nas configurações ordenado crescente, decrescente e aleatoriamente. Os dados de tempo de execução foram coletados e uma média dos mesmos calculados. A tabela a seguir apresenta estes dados:

Ordem Crescente				
	100	1000	10000	100000
Select	0	0.002	0.113	11.305
Insert	0	0	0	0
Bubble	0	0.002	0.105	10.385
Shell	0	0	0	0.003
Quick	0	0	0	0.004
Heap	0	0	0	0.011

Ordem Decrescente				
	100	1000	10000	100000
Select	0	0.002	0.105	10.600
Insert	0	0.002	0.105	10.700
Bubble	0	0.002	0.180	18.200
Shell	0	0	0.001	0.005
Quick	0	0	0.001	0.003
Heap	0	0	0.001	0.011

Ordem Aleatória				
	100	1000	10000	100000
Select	0	0.002	0.115	11.300
Insert	0	0.002	0.053	5.300
Bubble	0	0.002	0.230	25.000
Shell	0	0	0.001	0.002
Quick	0	0	0.001	0.012
Heap	0	0.001	0.001	0.015

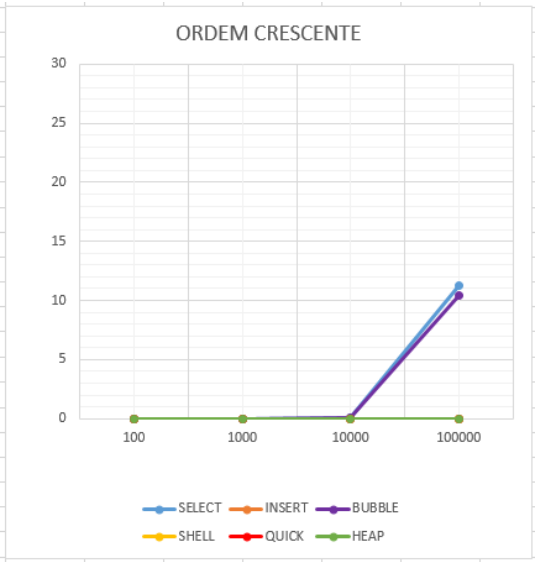
A partir destes dados foi possível verificar algumas características desses algoritmos para cada combinação tamanho x ordenação. Primeiro pode-se notar que arranjos de até 10000 itens não possuem um tempo de execução muito alto independente da ordenação inicial ou do algoritmo usado. Os arranjos de comparação então ficam sendo os de 100000 itens, onde começa a surgir grandes diferenças no tempo de execução.

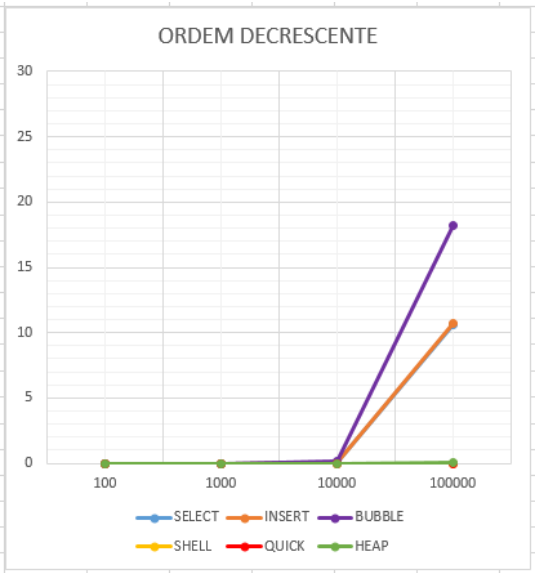
Em relação a arranjos com ordem crescente, ou seja, já ordenados, a melhor opção evidente é o *Insert Sort* que aproveita a ordenação já existente nos arranjos para ordenar apenas o que ainda não está em ordem. É possível verificar também que, analogamente, os algoritmos *Select Sort* e *Bubble Sort* não se aproveitam da já ordenação dos arranjos e, portanto, não se mostram boas opções para este tipo de conjunto de dados. Estes dois também apresentam um crescimento no tempo de execução numa escala bem maior que os *Shell*, *Quick* e *Heap Sort*.

Analisando os algoritmos totalmente desordenados, de maneira decrescente, os métodos simples *Select*, *Insert* e *Bubble Sort* todos apresentam um tempo de execução alto trabalhando com vetores grandes. Neste caso, o *Quick Sort* apresenta a menor média de tempo de execução.

Os arranjos ordenados aleatoriamente, representando os casos onde não se sabe o estado inicial dos dados, apresentam um tempo de execução ainda maior para os algoritmos simples, com destaque para a baixa eficiência do *Bubble Sort*. Os algoritmos mais complexos, de $O(n \log n)$, apresentam nesses casos um tempo de execução bem abaixo e com uma diferença desprezível entre eles. Apenas vale ressaltar o *Heap Sort* que possui o maior entre esses, talvez devido ao tempo gasto em montar o heap em cada execução.

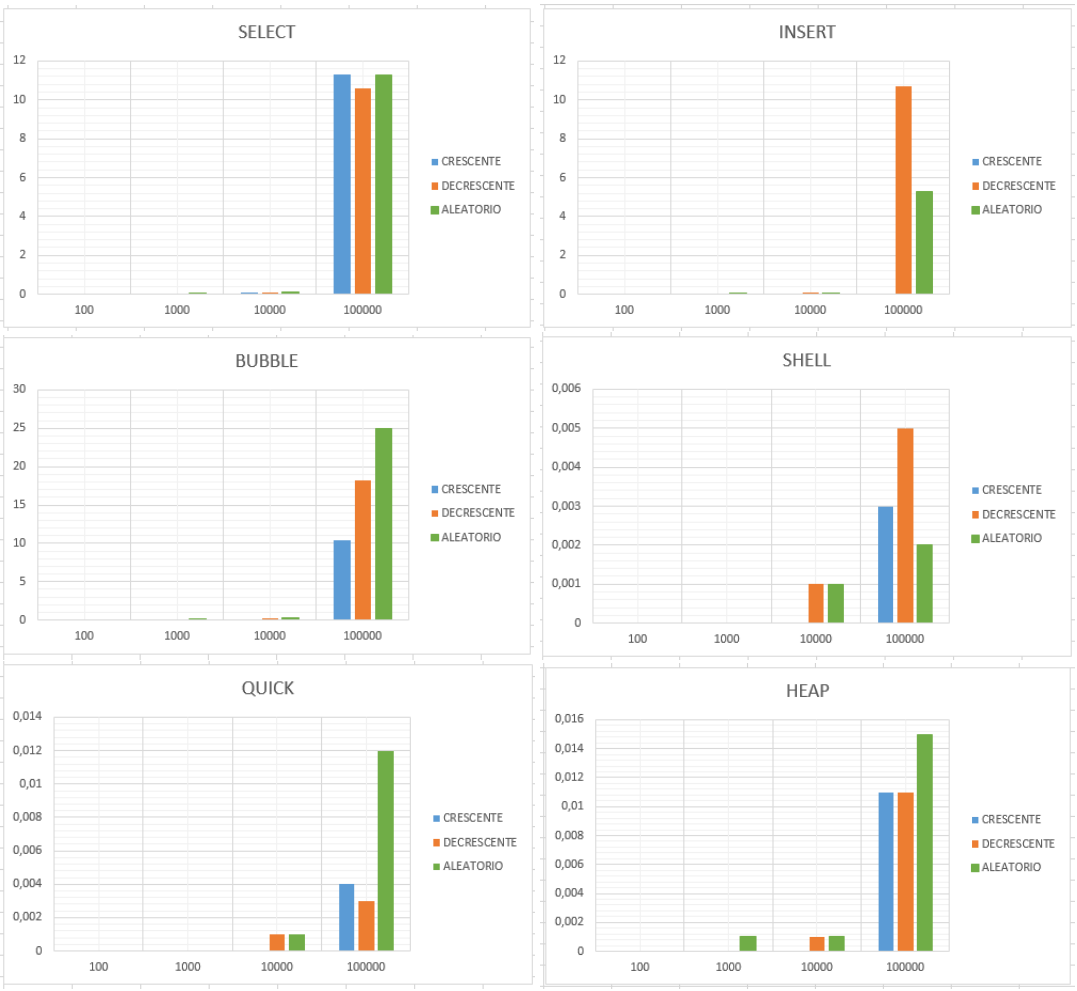
Os gráficos abaixo ilustram a taxa de crescimento dos algoritmos de ordenação e nestes é possível ver bem a ordem de complexidade.





Fazendo uma análise de cada um dos algoritmos separadamente podemos ver que, apesar de todos apresentarem crescimento conforme o tamanho dos arranjos, os algoritmos complexos tem um crescimento desprezível de apenas alguns mile segundos. Já os algoritmos simples, *Select* e *Bubble* apresentam um crescimento muito expressivo que pode ser prejudicial em um sistema muito grande e que demanda uma constante organização dos dados. Vale ressaltar também o comportamento do *Ìnsert Sort* quando o algoritmo já está ordenado. Ele se vê muito recomendado quando são necessárias diversas ordenações num mesmo arranjo ao longo da execução do programa visto que, apesar do tempo grande ao ordenar uma primeira vez, as outras execuções em cima do arranjo já parcialmente ordenado terá um custo baixo.

Seguem abaixo os gráficos “tamanho dos arranjos x tempo de execução” que ilustram essa análise:



5 Conclusão

A partir deste trabalho foi possível fazer uma comparação dos algoritmos de ordenação mais utilizados na computação e ver como é importante fazer, antes de ordenar qualquer arranjo, uma análise da situação dos dados no programa em questão para saber qual a melhor opção. Uma má escolha deste algoritmo, em um sistema grande e que demanda diversas ordenações recorrentes, pode gerar um tempo de execução muito grande e pode comprometer o funcionamento do sistema como um todo.