Algoritmos e Estruturas de Dados - Relatório 2
Comparação de Performances de Algoritmos de Ordenação

Filipe Pires 85122
João Alegria 85048

Introdução

Este relatório abarca não só a descrição da tarefa a nós proposta, da solução por nós implementada e dos resultados por nós obtidos, como também uma breve listagem dos maiores problemas de implementação que encontramos e uma conclusão sobre o trabalho no geral.

Tarefa / Objetivo

A tarefa nos proposta para este projeto resume-se na construção de estruturas de código que nos permita medir várias características dos algoritmos de ordenação mais conhecidos e calcular valores que nos ajudem a compreender o comportamento destes ao longo do tempo e consoante o tamanho da estrutura de dados a ser ordenada.

De entre os algoritmos, numeram-se: *BubbleSort*, *ShakerSort*, *InsertionSort*, *ShellSort*, *QuickSort*, *MergeSort*, *HeapSort*, *RankSort* e *SelectionSort*. De entre as características destes algoritmos, foi nos proposto medir os seus tempos de execução bem como o número de comparações feitas por cada um e nos pedido calcular os tempos médios de execução e os desvios padrões de cada algoritmo.

Tendo como base um "esqueleto" de código fornecido pelo professor, a tarefa não era nada mais nada menos que recolher a informação procurada e apresentar os resultados obtidos.

Solução / Implementação

Para a implementação da medida dos tempos de execução dos algoritmos de ordenação, começámos por adicionar ao programa a função *elapsed_time()*, que permite inicializar um contador de tempo e guardar as suas medições em qualquer ponto do código. Desta forma, sempre que uma função de ordenação é chamada, o contador serve como cronómetro para o seu tempo de execução. Com os tempos recolhidos de cada função podemos proceder aos cálculos das respetivas médias e desvios padrão.

Abaixo apresentamos os gráficos com a evolução destas médias e desvios padrão consoante o tamanho dos *arrays* ordenados. Para que a leitura dos gráficos seja mais fácil, deixamos aqui os nomes dos algoritmos de ordenação ordenados crescentemente por tempos de execução: *QuickSort*, *MergeSort*, *ShellSort*, *HeapSort*, *InsertionSort*, *RankSort*, *ShakerSort*, *BubbleSort* e finalmente *SelectionSort*.

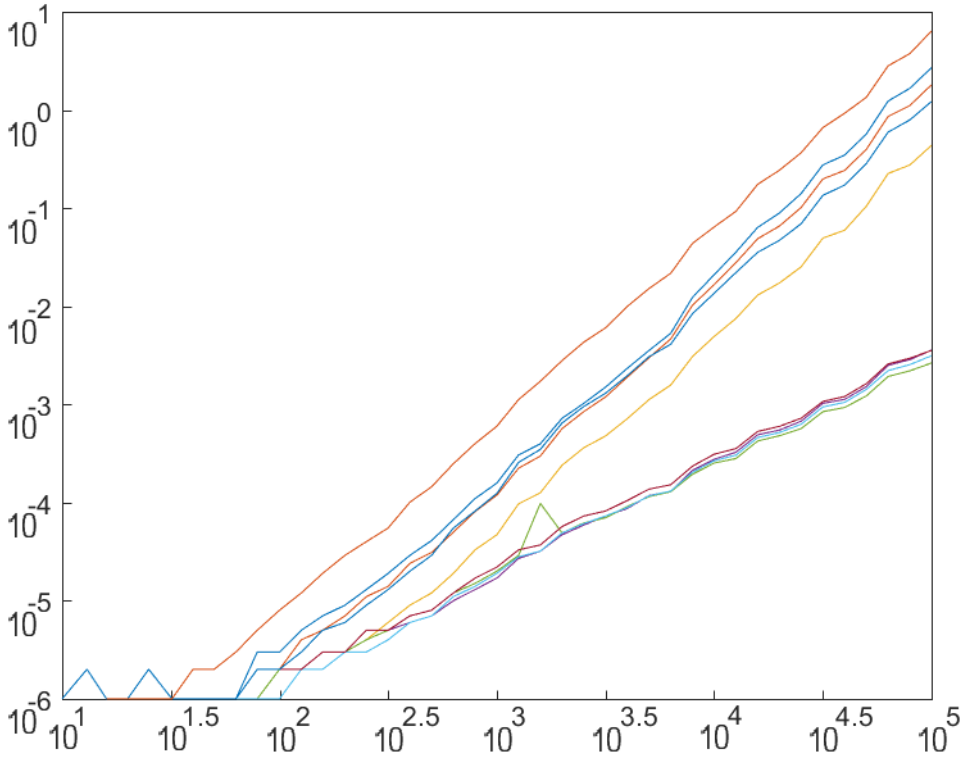

Figura 1: Gráfico da evolução dos tempos médios. x = tamanho dos *arrays*. y = tempo (segundos)

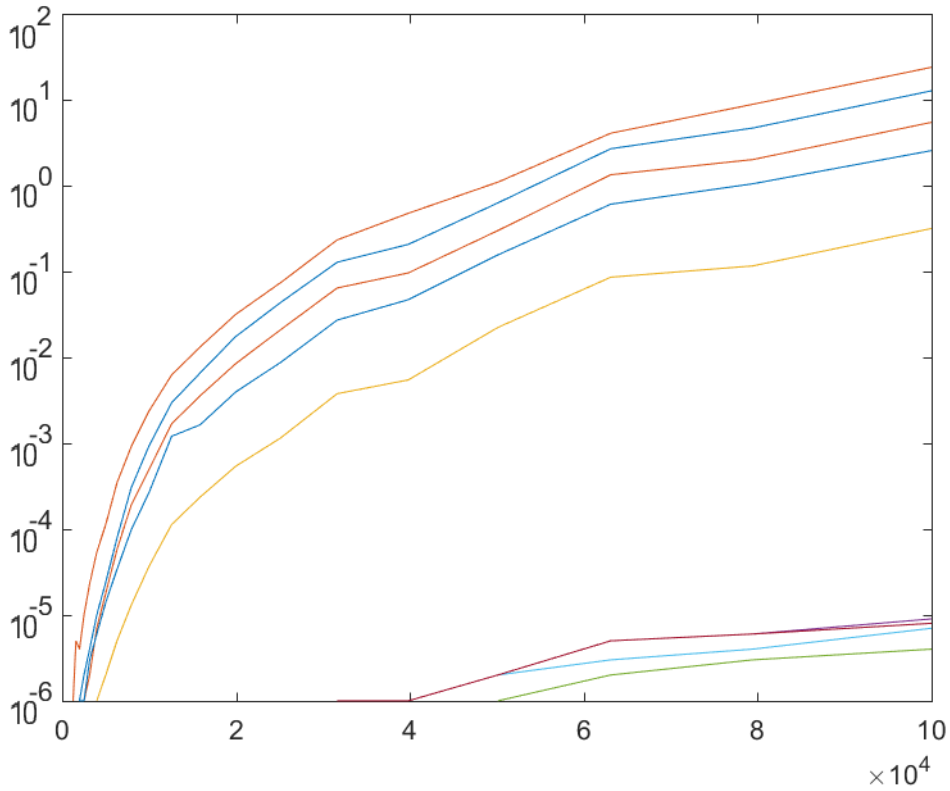

Figura 2: Gráfico da evolução dos desvios padrão. x = tamanho dos *arrays*. y = desvio padrão (segundos)

Já para a implementação da contagem de comparações foi feita através de uma estrutura (de nome *counts*), cujo primeiro atributo corresponde ao contador de comparações de cada algoritmo de ordenação e o segundo atributo corresponde ao contador de trocas de elementos feitas por alguns dos algoritmos.

As alterações necessárias a serem feitas no código do professor foram, portanto, alterar o valor de retorno de cada função para *counts*, coletar os dados obtidos através destes valores e guardar num ficheiro os resultados. O gráfico abaixo apresentado representa a soma do total de contagens de comparações feitas por cada algoritmo para *arrays* que cresciam logarítmicamente de 10 a 100000 elementos. Aqui vê-se uma relação de proporcionalidade direta entre o número de comparações e o tempo de execução.

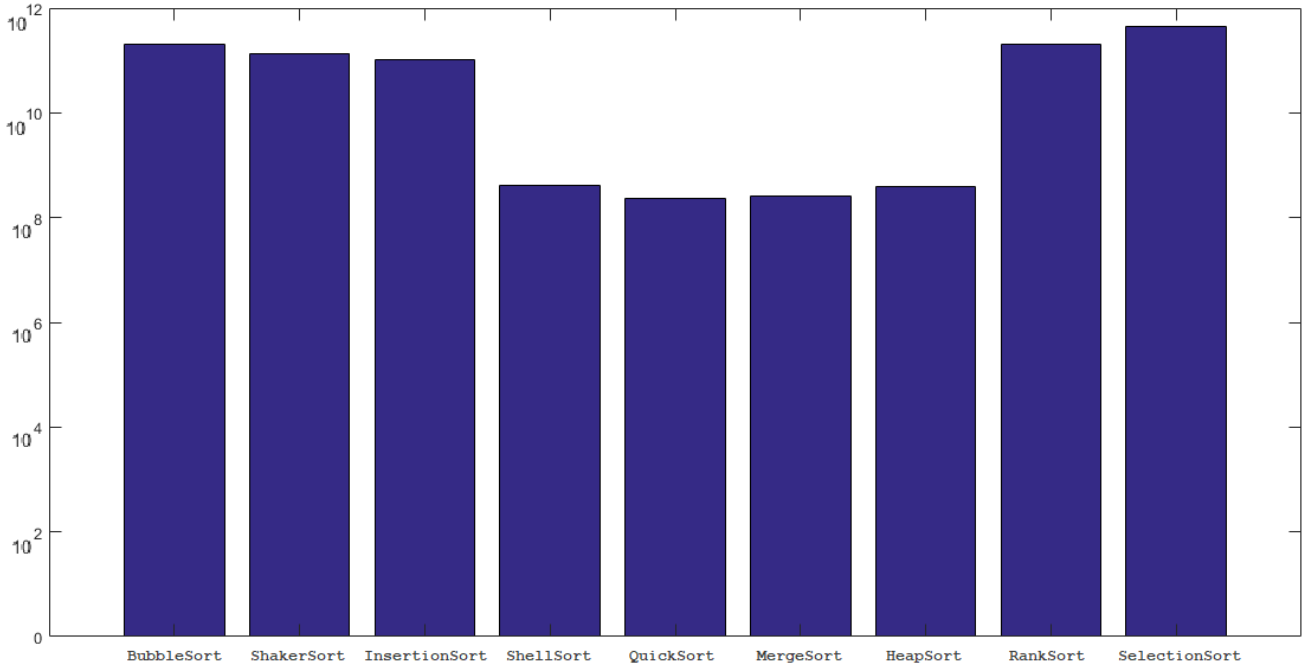

Figura 3: gráfico de barras do total de comparações feitas pelos algoritmos.
x = algoritmos de ordenação. y = nº de comparações

O segundo atributo da estrutura *counts* foi então utilizado após as implementações anteriores serem completadas. A ideia posta em prática foi a contagem do número de trocas feitas por três algoritmos em específico: *BubbleSort*, *QuickSort* e *SelectionSort*. A criação do contador foi relativamente simples e, como a estrutura já existia, não foi necessário alterar mais linha de código nenhuma. Abaixo deixamos o gráfico dos resultados obtidos por cada algoritmo.

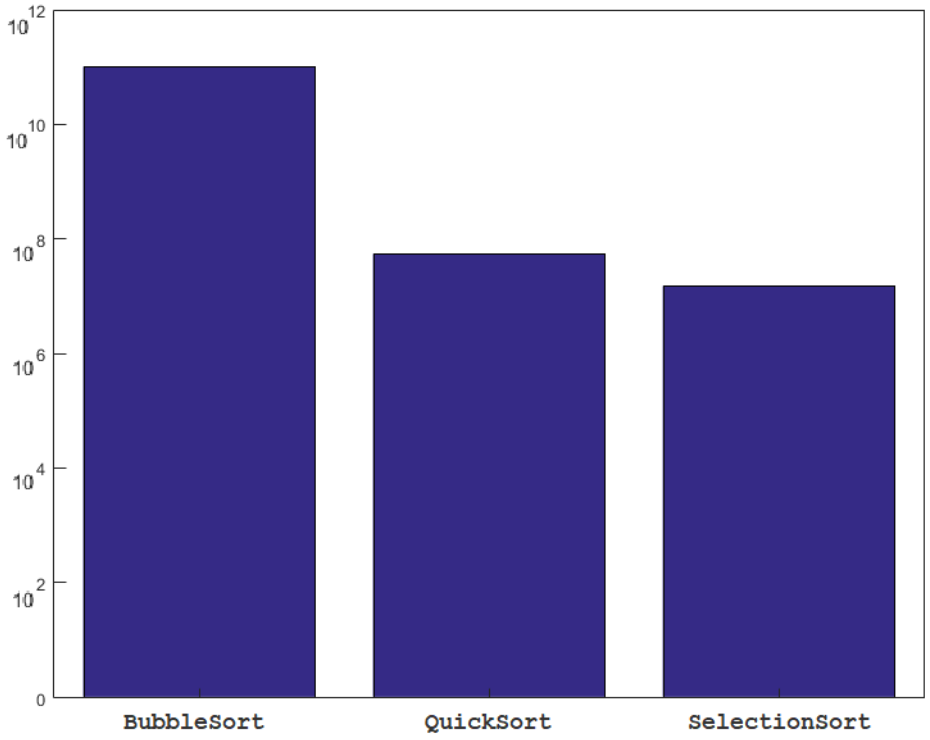

Figura 3: gráfico de barras do total de comparações feitas pelos algoritmos.
x = algoritmos de ordenação. y = nº de comparações

Problemas / Dificuldades

Ao contrário do primeiro projeto, a medição de performances dos algoritmos de ordenação trouxe-nos poucos problemas a nível de implementação. As tarefas eram bastante lineares e os poucos impasses que ocorriam eram na sua maioria distrações e erros de sintaxe.

A implementação mais difícil de completar foi o *array* bidimensional que continha as médias dos tempos de execução de cada algoritmo. Devido a chamadas a zonas de memória inacessíveis ou utilizações de ponteiros de forma errada, o tempo gasto no desenvolvimento da solução aumentou um pouco devido a esta estrutura. Ainda assim, com perseverança e calma, todos os problemas foram resolvidos e podemos garantir que o programa está funcional e completo.

Conclusão

Com este projeto aprendemos que alguns algoritmos de ordenação utilizam estruturas de dados auxiliares de forma a acelerarem as suas performances e concluímos que na maioria dos casos o algoritmo de ordenação mais eficaz é o *QuickSort* (ainda que tenhamos utilizado uma implementação deste algoritmo diferente da fornecida pelo professor).

A nível do desempenho dos elementos do grupo, podemos concluir que trabalhamos bem como par e mantemos uma comunicação constante de forma a estarmos sempre a par do ponto de situação do trabalho, resultando num bom desempenho neste tipo de tarefas.

Quanto à solução por nós implementada, podemos concluir que todas as "sub-tarefas" propostas foram completadas, assim como algumas extra, e pensamos que a forma como estão construídas as soluções é a mais eficaz possível.

Fontes de Pesquisa:

https://stackoverflow.com/
https://www.tutorialspoint.com/cprogramming/index.htm
https://www.tutorialspoint.com/c_standard_library/index.htm