Trabalho Prático 01

Thiago Dourado de Andrade

091633

Este trabalho é um exercício prático do conteúdo exposto em sala de aula na matéria de Análise de Algoritmos, pelo professor Márcio Palheta. O trabalho tem como objetivo a implementação e análise de três algoritmos de ordenação, heapsort, mergesort e quicksort. Para observar as diferenças nos algoritmos serão utilizados tamanhos de vetores de 10.000, 100.000, 1.000.000, 10.000.000 e 100.000.000 com números aleatórios entre 0 e 65.535.

O trabalho foi implementado em C, com cada algoritmo de ordenação em um arquivo-fonte diferente e um único arquivo-fonte para realizar a ordenação. O programa que realiza a ordenação recebe como parâmetros o algoritmo a ser usado para ordenação, heap, merge e quick, o arquivo a ser ordenado contendo um número por linha e opcionalmente um arquivo para ser escrito o arquivo ordenado.

Invocação:

sort {heap | merge | quick} <arquivo entrada> [<arquivo saída>]

O programa `sort` escreve na saída padrão o tempo de ordenação no formato MM:SS:mmm,000, usando a função gettimeofday() padrão de sistemas POSIX, e na saída de erro escreve mensagens mais compreensíveis, informando quando terminou de ler o arquivo de entrada.

Além do programa principal existem também programas auxiliares para realização dos testes. O programa `gerarNumeros` implementado em C e, com números aleatórios criptograficamente fortes, em Go, o programa `media` e o Script `executaTestes.sh`. Para gerar todos os arquivos necessários para replicação dos testes basta rodar o comando `make`. Para apagar todos os executáveis gerados, basta rodar `make remove`.

O programa `gerarNumeros` recebe opcionalmente 2 parâmetros, a quantidade de números a ser gerado e o arquivo onde esses números devem ser escritos. Caso não seja passado o arquivo os números serão escritos em *numeros*, e caso não seja passado a quantidade de números a ser gerado, serão gerados 10.000 números. Para usar a versão em Go, é preciso ter o compilador de Go instalado, `sudo apt-get install golang` no Ubuntu, e rodar `go build gerarNumeros.go`.

Invocação:

gerarNumeros [quantidade [<arquivo saída>]]

O programa `media` lê da entrada padrão processa os resultados dos testes e escreve na saída padrão. É usado para calcular a média dos tempos. Espera como entrada os resultados formatados como serão escritos pelo script `executaTestes.sh`.

Invocação:

media <<arquivo resultados> ><arquivo média>

O script `executaTestes.sh` recebe o número do teste e um parâmetro opcional para limpar o arquivo de resultados. O número do teste é a potência de 10 necessária que multiplicando 1.000 obtém o tamanho desejado, 1 para 10.000, 2 para 100.000 e assim por diante.

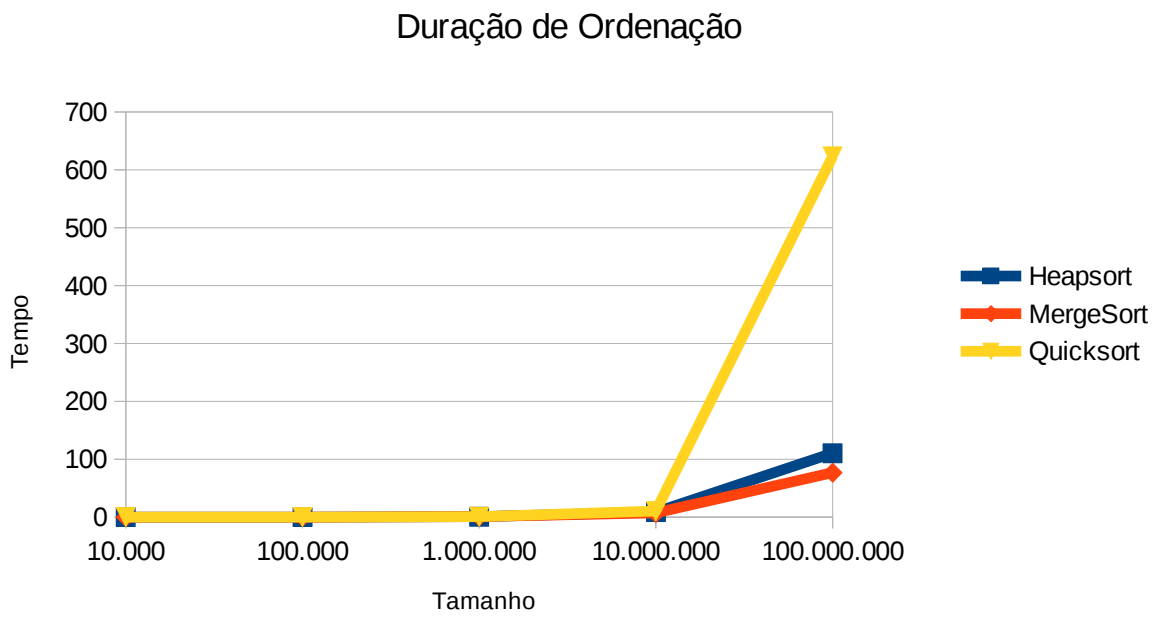
Invocação:

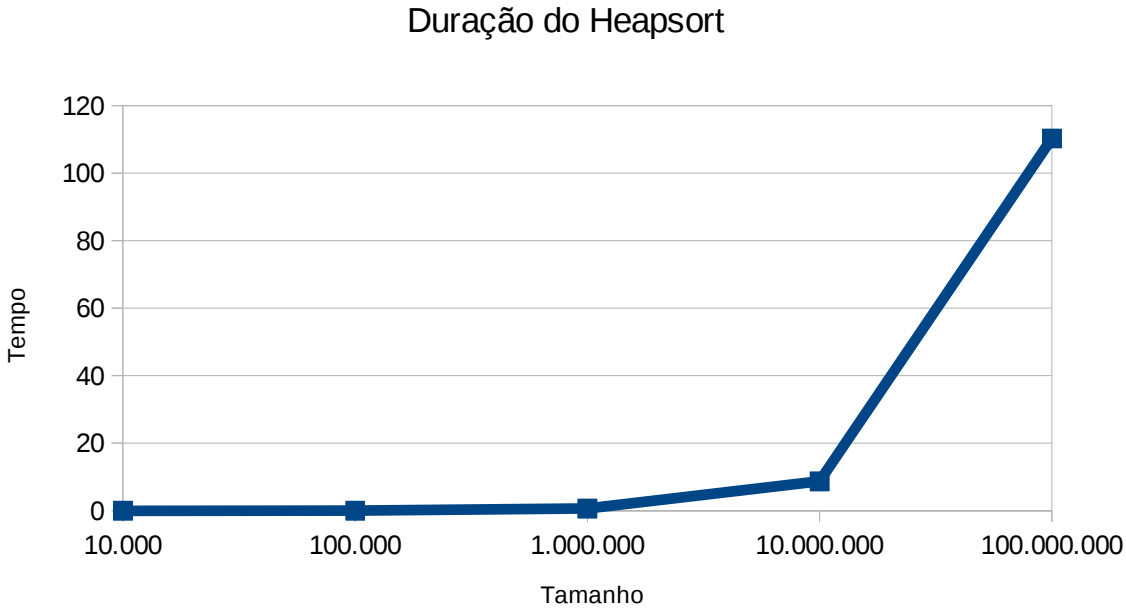
executaTestes.sh numero {novo}

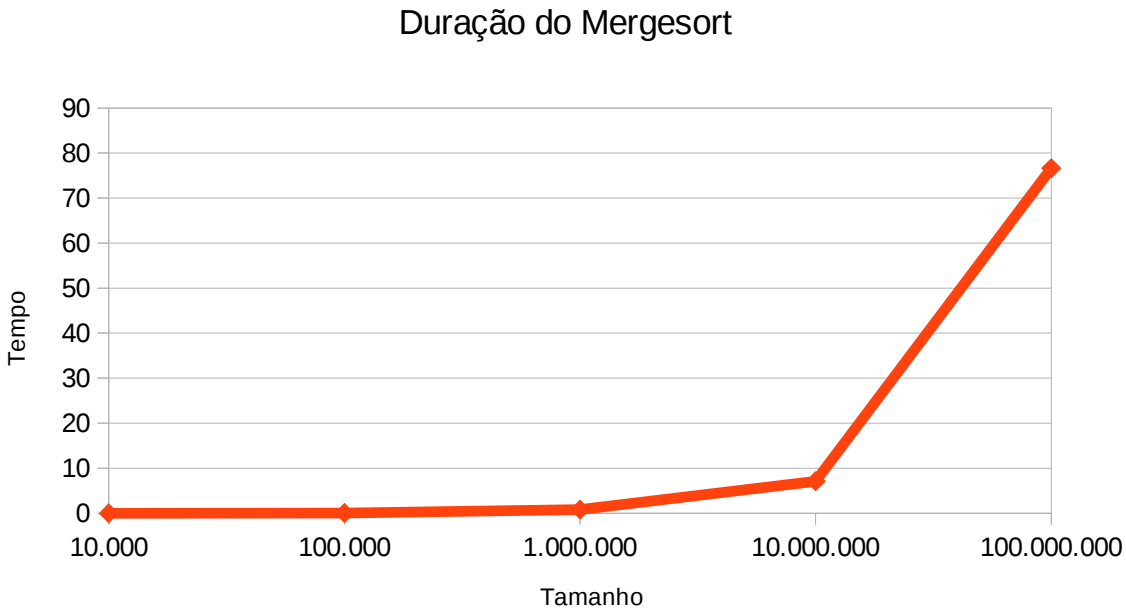
Cada teste foi executado 50 vezes, com exceção dos testes para o tamanho de 100.000.000, que foram executados 5 vezes cada. Para isso o script modificado para iterar de 1 a 5 ao invés de 1 a 50, como está. Os resultados das execuções são escritas nos arquivos testeHeap, testeMerge e testeQuick. O resultados obtidos durante o experimento ainda estão disponíveis nestes arquivos. As médias finais se encontram nos arquivos resultadosHeap, resultadosMerge e resultadosQuick.

A seguir encontra-se uma tabela com todos os tempos de ordenação em segundos e os gráficos gerados a partir desta tabela, comparando cada algoritmo. Os experimentos foram realizados usando a versão criptograficamente forte dos números aleatórios, a versão escrita em Go.

Método x Tamanho	**10.000**	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**
Heapsort	00,003425	00,042394	00,654321	08,710898	110,251589
MergeSort	00,004677	00,054280	00,821708	07,109317	76,635341
Quicksort	00,003731	00,043078	00,709107	10,425797	623,213592




Duração do Heapsort
120
100
80
60
40
20
0
Tempo
10.000
100.000
1.000.000
10.000.000
100.000.000
Tamanho


Duração do Mergesort
90
80
70
60
50
40
30
20
10
0
Tempo
10.000
100.000
1.000.000
10.000.000
100.000.000
Tamanho

Pelos dados observados com o experimento, e lembrando que os tempos estão em segundos, podemos perceber que até a quantidade de 10.000.000 de números não existe uma diferença significativa entre os três algoritmos, mas é possível observar algumas tendências discutidas a seguir.

Até 1.000.000 de números os algoritmos mantiveram a ordem de eficiência com o heapsort sendo o mais eficiente seguido do quicksort por último o mergesort. Com 10.000.000 de números, apesar da diferença ser de no máximo 3 segundos, a tendência muda e o mergesort ordena de forma mais eficiente seguido do heapsort e então o quicksort.

Com 100.000.000 de números, os tempos de ordenação se tornam maiores e as diferenças nos tempos mais drásticas. O mergesort é claramente o mais rápido de todos ordenando em 1 minuto e 16 segundos em média, com o heapsort em seguida com um tempo em torno de 144% o tempo do mergesort. Já o quicksort não fez jus ao seu nome e executou as ordenações em uma média de 10 minutos e 23 segundos, em torno de 819% o tempo do mergesort.

Baseando-se nos limites superiores calculados em sala de aula, aparentemente o quicksort está rodando em um tempo próximo ao quadrático enquanto o mergesort e o heapsort estão executando em tempo logarítmico, o que explicaria a grande diferença entre o tempo do quicksort e os outros. Para identificar o motivo disso seria necessário registrar as listas de números desordenadas para analisá-las.

Outro fator que pode estar afetando o tempo de execução do quicksort é a escolha do pivô. Em minha implementação usei um pivô aleatório, o que teoricamente tem a melhor chance de ser eficiente não assumindo mais nenhuma informação sobre a entrada. Afinal, como os testes não cobriram mais cenários, não é possível determinar o motivo dessa grande diferença entre os tempos no último teste.