Computação Aplicada - Lista 05

Rodrigo Takeshi Seo

Junho 2017

1 Exercício 1

Os testes de execução dos algoritmos de ordenação foram separados em cenários para que seja possível realizar uma melhor comparação dos resultados. Os resultados apresentados nos subcapítulos abaixo são o resultado da média de 100 medições de tempo dentro dos seguintes cenários:

Tabela 1: Descrição dos Cenários de Testes

Cenário 1	10000 inteiros gerados de forma aleatória com valores distribuídos entre 10^0 e 10^6
Cenário 2	100000 inteiros gerados de forma aleatória com valores distribuídos entre 10^0 e 10^6
Cenário 3	10000 inteiros ordenados [0, 1, 2, 3...9999]
Cenário 4	10000 inteiros inversamente ordenados [9999, 9998, 9997 ... 0]

Estrutura dos Arquivos

O arquivo *"sorts.cpp"* contém as implementações dos algoritmos de ordenação;

No arquivo *"exec.cpp"* está contido a função de geração de números aleatórios e a rotina de execução da bateria de testes *exec(double *v)* o vetor v[] esperado nesta função contém o somatório dos tempos de execução dos algoritmos em cada cenário. Cada posição do vetor contém:

- De 0 a 8 somatório de tempos do cenário 1;
- De 9 a 17 somatório de tempos do cenário 2;
- De 18 a 26 somatório de tempos do cenário 3;
- De 27 a 35 somatório de tempos do cenário 4;

No arquivo *"main.cpp"* é definido a variável *max* que representa a quantidade de vezes que as medições serão executadas. Após realizadas as execuções, os valores somados no vetor *v*, parametro da função exec, são divididos pela

variável max, resultando a média de tempo de execução de cada algoritmo em cada cenário.

Para garantir que a comparação entre os algoritmos seja correta, foram criados arrays contendo os mesmos valores para todos os algoritmos, como mostra a Figura 1:

```
114     //Cenário 2
115
116     int vSelection2[100000];
117     int vInsert2[100000];
118     int vMerge2[100000];
119     int vQuickSort2[100000];
120     int vShellSort2[100000];
121     int vHeapSort2[100001];
122     int vStlSort2[100000];
123     int vStlStableSort2[100000];
124     int vStlSortHeap2[100000];
125     vHeapSort[0] = -1;
126
127      for(int k = 0; k <= 99999; k++){
128         int n = randomNumber(1, 1000000);
129         vSelection2[k] = n;
130         vInsert2[k] = n;
131         vMerge2[k] = n;
132         vQuickSort2[k] = n;
133         vShellSort2[k] = n;
134         vHeapSort2[k+1] = n;
135         vStlSort2[k] = n;
136         vStlStableSort2[k] = n;
137         vStlSortHeap2[k] = n;
138     }
```

Figura 1: Criação e Preenchimento dos Vetores

Observações:

- Utilizando a estrutura vector da STL, o tempo de execução dos algoritmos aumentava, por este motivo decidiu-se a utilização de arrays simples no experimento.
- Apenas o vetor utilizado no HeapSort é inicializado de forma diferente, pois o algoritmo considera os valores alocados nas posições 1 a n.
- Os melhores resultados em cada cenário são apresentados em negrito nas tabelas.

1.1 Resultados do Cenário 1

Tabela 2: Tempo Médio de Execução dos Algoritmos de Ordenação - Cenário 2

Algoritmo	**Tempo(s)**	**Tempo(ms)**
Seleção	0,3341460726	334,1460726
Inserção	0,1919092898	191,9092898
Shell	0,0043012227	4,3012227
Heap	**0,0035223325**	**3,5223325**
Merge	0,0043455073	4,3455073
Quick	0,0040227276	4,0227276
STL Sort	0,0043084545	4,3084545
STL Stable Sort	0,0045750166	4,5750166
STL Sort Heap	0,0059264662	5,9264662

Neste cenário de execução é possível observar que o Heap Sort apresentou melhor performance dentre todos os algoritmos. Neste cenário, não foi possível identificar muito bem as diferenças entre as estratégias de ordenação, pois muitos algoritmos apresentaram tempos equivalentes.

Os algoritmos com complexidade quadrática (Seleção e inserção), como esperado para este cenário, apresentaram tempo bem superior aos outros.

A Figura 2 apresenta o gráfico da média de tempo da execução dos melhores algoritmos do cenário 1, em milisegundos.

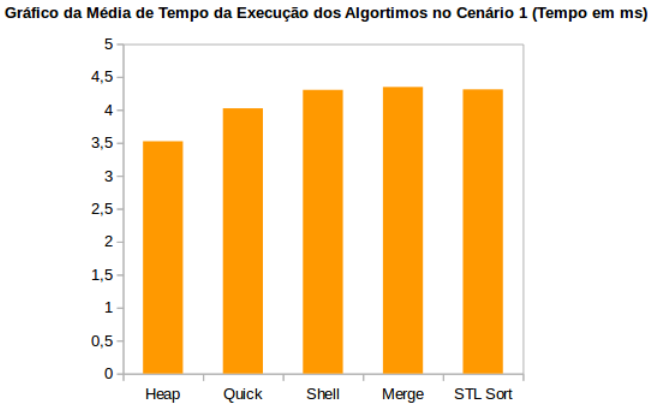

Figura 2: Gráfico da Média de Tempo da Execução do Cenário de Testes 2

1.2 Resultados do Cenário 2

Tabela 3: Tempo Médio de Execução dos Algoritmos de Ordenação - Cenário 2

Algoritmo	**Tempo(s)**	**Tempo(ms)**
Seleção	32,1340560303	32134,0560303
Inserção	19,7677186486	19767,7186486
Shell	0,0644238878	64,4238878
Heap	0,0503586806	50,3586806
Merge	0,0547829869	54,7829869
Quick	**0,0370751853**	**37,0751853**
STL Sort	0,0545867441	54,5867441
STL Stable Sort	0,0579146909	57,9146909
STL Sort Heap	0,076256582	76,256582

Neste cenário é interessante observar a performance do QuickSort, pois o mesmo apresentou tempo de execução proporcionalmente melhor do que apresentado no cenário anterior, enquanto os outros algoritmos que apresentaram bons resultados no cenário 1 não mantiveram a performance com o vetor 10 vezes maior. Sendo assim, é possível concluir que o QuickSort é mais viável para vetores maiores gerados com valores aleatórios.

Como esperado dos algoritmos de complexidade quadrática, o tempo de execução deles cresceu proporcionalmente ao número de operações.

A Figura 3 apresenta o gráfico da média de tempo de execução dos melhores algoritmos do cenário 2, em milisegundos.

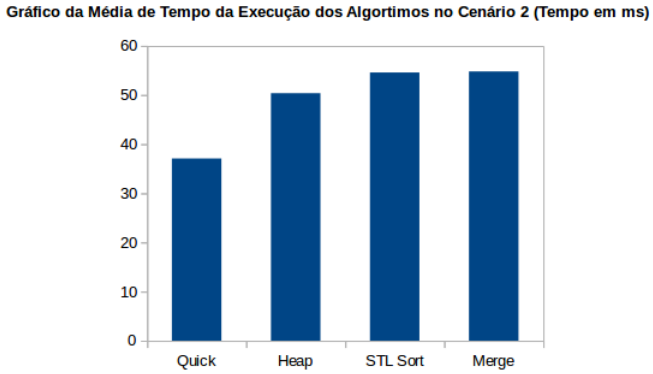

Figura 3: Gráfico da Média de Tempo da Execução do Cenário de Testes 2

1.3 Resultados do Cenário 3

Tabela 4: Tempo Médio de Execução dos Algoritmos de Ordenação - Cenário 3

Algoritmo	**Tempo(s)**	**Tempo(ms)**
Seleção	0,3556799457	355,6799457
Inserção	**0,0001238711**	**0,1238711**
Shell	0,0009541578	0,9541578
Heap	0,0028531451	2,8531451
Merge	0,0027061126	2,7061126
Quick	0,3036713792	303,6713792
STL Sort	0,002309535	2,309535
STL Stable Sort	0,0019303126	1,9303126
STL Sort Heap	0,0052069054	5,2069054

Neste cenário de vetores ordenados, fica claro a diferença que a estratégia de ordenação é extremamente importante para a melhor performance do algoritmo. O algoritmo de inserção que possui formalmente complexidade quadrática, apresenta o melhor tempo devido a sua estratégia, que neste cenário (melhor cenário possível) possui comportamento N. O ShellSort que possui estratégia parecida aparece em segundo lugar neste cenário.

O QuickSort é o segundo pior algoritmo neste cenário devido a estratégia de separação do vetor, no algoritmo implementado neste experimento, o pivo é sempre a ultima posição, tornando o algoritmo praticamente quadrático.

A Figura 4 apresenta o gráfico da média de tempo de execução dos melhores algoritmos do cenário 3, em milisegundos.

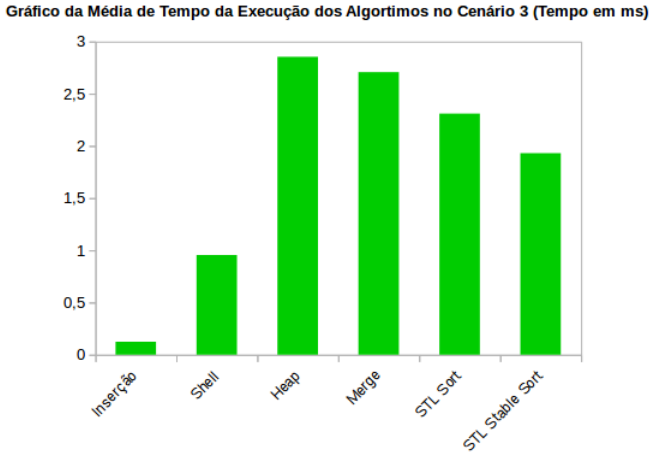

Figura 4: Gráfico da Média de Tempo da Execução do Cenário de Testes 3

1.4 Resultados do Cenário 4

Tabela 5: Tempo Médio de Execução dos Algoritmos de Ordenação - Cenário 4

Algoritmo	**Tempo(s)**	**Tempo(ms)**
Seleção	0,2321771868	232,1771868
Inserção	0,2735442524	273,5442524
Shell	**0,0010862947**	**1,0862947**
Heap	0,0020317913	2,0317913
Merge	0,0019143727	1,9143727
Quick	0,2133771455	213,3771455
STL Sort	0,0012369789	1,2369789
STL Stable Sort	0,0018157751	1,8157751
STL Sort Heap	0,0037658435	3,7658435

Neste cenário, ao contrário do anterior, a ordenação por inserção apresentou o pior resultado, devido a sua estratégia de ordenação, este cenário é o pior possível para o algoritmo, pois a cada elemento, ele deverá empurrar todos os outros elementos já alocados para a final do vetor.

O ShellSort apresentou o melhor resultado neste cenário, algo que é contraintuitivo, já que o Shell adota estratégia de ordenação bem parecida com a ordenação por inserção.

A Figura 5 apresenta o gráafico da média de tempo de execução dos melhores algoritmos do cenário 4, em milisegundos.

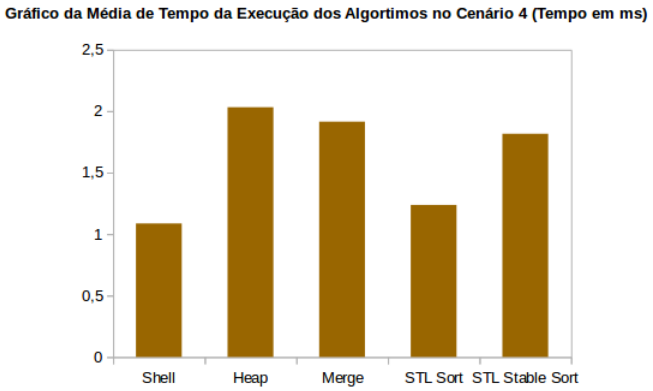

Figura 5: Gráfico da Média de Tempo da Execução do Cenário de Testes 4

1.5 Considerações Finais

O primeiro aspecto a se observar é que alguns algoritmos apresentaram boa performance em todos os cenários de teste, podendo ser uma boa opção de

escolha para implementação em casos reais, entre esses, cita-se:

- Heap Sort;
- Merge Sort;
- Stl Sort;
- Stl Stable Sort; e
- Stl Sort Heap

Neste experimento, pede-se sugestões de melhorias nos algoritmos clássicos de ordenação, porém, dependendo da alteração realizada no algoritmo, pode-se descaracterizar a estratégia de ordenação que o nomeia.

Uma alteração positiva que é possível de realizar sem alterar a caracteristica do algoritmo é o método de escolha do pivô no QuickSort, há alguns exemplos na literatura onde utilizam-se n/2 como pivô, esta estratégia reduziria o tempo de ordenação para vetores quase, ou totalmente ordenados.