Gustavo Barbaro de Oliveira
Vinícius Rodrigues Miguel

Relatório do Primeiro Trabalho em Grupo
Estudo de Algoritmos de Ordenação

São José dos Campos - Brasil

Abril de 2019

Gustavo Barbaro de Oliveira
Vinícius Rodrigues Miguel

Relatório do Primeiro Trabalho em Grupo
Estudo de Algoritmos de Ordenação

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Algortimos e Estruturas de Dados II.

Docente: Prof. Dr. Reginaldo Massanobu Kuroshu

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Abril de 2019

Lista de tabelas

Sumário

1 Geração de Conjuntos de Entrada

Para que fosse possível que todos os algoritmos recebessem um mesmo *input*, foi projetado um *script* na linguagem *Go*. De início, este gerava arquivos com números aleatórios no intervalo $[-2.000.000.000, 2.000.000.000]$. Percebe-se, porém, que este intervalo traria uma performance péssima para os algoritmos com complexidade assintótica dependente de k como o Radix Sort e o Counting Sort.

A fim de reduzir a amplitude dos números gerados, foi-se reprojetado o *script* para que este gerasse números nos seguintes intervalos, sendo n a quantidade de números a serem gerados:

1. Para gerar números aleatórios sem ordem, adota-se o intervalo de $[0, \lfloor n/ln(ln(n)) \rfloor]$
2. Para a geração de números aleatórios ambos em ordem crescente e decrescente, adota-se o intervalo de $[2, \lfloor log2(log2(log2(n))) \rfloor]$. A geração de números em ordem decrescente tem sempre como valor máximo n/2 ou n/2 - 1.

2 Grupo I: Selection, Insertion

2.1 Insertion Sort

Considerado como um dos algoritmos de ordenação mais simples, o Insertion sort se baseia na ideia de inserção de um elemento em um subvetor ordenado, de maneia análoga à maneira pela qual cartas são comumente ordenadas. O algoritmo é especialmente útil quando tem-se um vetor semi-ordenado.

2.1.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 1.

Insertion	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	0.035000 ms	4.811000 ms	3.576000 ms
10.000	0.140000 ms	464.450000 ms	231.219000 ms
100.000	1.338000 ms	46641.708000 ms	23348.745000 ms
1.000.000	13.318000 ms	43934837.230000 ms	22344492.745000 ms

Tabela 1 – Insertion Sort com os tempos de execução

Pode-se perceber o comportamento quadrático da Ordenação por Inserção, isto é, o tempo de execução cresce de maneira sempre quadrática ao tamanho de entrada, salvo nos casos onde os elementos já estão ordenados crescentemente.

2.2 Selection Sort

A ordenação por seleção consiste em percorrer o vetor, encontrar seu menor valor, substituí-lo na posição inicial do vetor, após feito isso, incrementa-se uma posição e o o laço inicia novamente procurando o o segundo menor valor, para colocar na segunda posição do vetor, e assim sucessivamente.

2.2.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 2.

Selection	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	4.2215 ms	11.6235 ms	10.138 ms
10.000	370.118000 ms	1106.134000 ms	772.293 ms
100.000	37280.069000 ms	113334.024000 ms	75700.552000 ms
1.000.000	Em teoria, levará mais de 2 horas		

Tabela 2 – Selection Sort com os tempos de execução

Assim como o Insertion, o Selection apresenta um crescimento quadrático ainda mais grave, tendo ainda a desvantagem de continuar sendo $\mathcal{O}(n^2)$ até para o caso de vetor já ordenado.

3 Grupo II: Quick, Merge

3.1 Quick Sort

Um dos algoritmos normalmente eficiente, o Quick Sort utiliza a técnica da divisão e conquista para rearranjar as chaves do vetor de modo que as chaves menores precedam as chaves maiores. Em seguida ele ordena as duas sub-listas de chaves menores e maiores recursivamente até que a lista fique ordenada.

3.1.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 3.

Quick	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	9.920000 ms.	4.578000 ms.	0.225000 ms.
10.000	818.878000 ms.	370.852000 ms.	2.377000 ms.
100.000	83050.890000 ms.	35541.581000 ms.	30.312000 ms.
1.000.000	Demora muito.	Demora muito.	357.496000 ms.

Tabela 3 – Quick Sort com os tempos de execução.

O algoritmo de Ordenação por Partição demonstra um funcionamento rápido para valores desordenados, levando, para estes casos, cerca de metade do tempo levado pelo Merge Sort para os mesmos valores. O algoritmo, no entanto, aparenta funcionamento péssimo quando os valores estão ordenados, seja crescente ou decrescentemente, apresentando nestes casos um funcionamento ainda pior em relação ao Insertion Sort e Selection Sort.

3.2 Merge Sort

O merge sort é um outro exemplo de algoritmo bastante otimizado que se utiliza da técnica de dividir para conquistar. De forma bastante semelhante ao quick sort, utiliza chamadas recursivas para conseguir particionar um problema grande em outros pequenos, de fácil solução.

Os três passos fundamentais desse algoritmo são :

- Dividir: Calcula o ponto médio do vetor para fazer a primeira divisão.
- Conquistar: Utilizando chamadas recursivas, resolve dois subproblemas cada um de tamanho $\frac{N}{2}$

- Combinar: Unir os dois sub arranjos em um único conjunto ordenado.

3.2.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 4.

Merge	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	0.425000 ms.	0.578 ms.	0.835000 ms.
10.000	3.765000 ms.	4.050200 ms.	4.472000 ms.
100.000	42.759000 ms.	43.512000 ms.	52.329000 ms.
1.000.000	490.260000 ms.	496.719000 ms.	602.378000 ms.

Tabela 4 – Merge Sort com os tempos de execução.

Embora tenha um funcionamento de Caso Médio pior em comparação ao Quick Sort, o Merge apresenta maior equilíbrio no sentido de que este funciona de maneira similar para qualquer tipo de valores de entrada, estejam ele caóticos, quase-ordenados ou ordenados, diferentemente do Quick.

4 Grupo III: Counting, Radix

4.1 Counting Sort

O Counting Sort é um algoritmo não-comparativo, cujo funcionamento resumido se dá por contar o número de itens distintos e então calculando a posição final destes na ordem de saída. O Counting é eficiente quando o alcance (*range*) dos dados de entrada não é significantemente maior do que o número de itens a serem ordenados. Ele costuma ser utilizado como sub-rotina para outros algoritmos de ordenação, tais como o Radix Sort.

4.1.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 5.

Counting	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	0.034000 ms.	0.047000 ms.	0.056000 ms.
10.000	0.425000 ms.	0.494000 ms.	0.385000 ms.
100.000	4.312000 ms.	3.210000 ms.	5.210000 ms
1.000.000	32.421000 ms.	28.201000 ms.	27.581000 ms.

Tabela 5 – Counting Sort com os tempos de execução.

A natureza de ordem linear do Counting Sort se demonstrou forte nas simulações feitas, visto que este obteve resultados muito melhores do que obtiveram os algoritmos comparativos testados até o momento, sendo eficiente até para os casos-teste de um milhão de valores, visto que o *script* em *Go* cria valores máximos em torno de n/2 (k = 500.000 quando n = 1.000.000).

4.2 Radix Sort

O Counting sort é bem eficiente para ordenar vetores cujo tamanho, `n`, não é muito menor que o tamanho máximo do vetor, `k`. O algoritmo Radix Sort utiliza diversas chamadas de Counting-Sort para permitir um alcance (*range*) maior de valores máximos. O Radix Sort ordena inteiros de w-bits ao chamar w/d chamadas de Counting Sort para ordenar esses inteiros, d bits de cada vez.

4.2.1 Resultados

Após as execuções, foram obtidos os seguintes resultados mostrados na Tabela 6.

Radix	Tempo		
Entradas	Crescente	Decrescente	Aleatório
1.000	0.040000 ms.	0.040000 ms.	0.042000 ms
10.000	0.455000 ms.	0.344000 ms.	0.345000 ms.
100.000	3.612000 ms.	3.514000 ms.	4.317000 ms.
1.000.000	35.995000 ms.	34.837000 ms.	85.824000 ms.

Tabela 6 – Radix Sort com os tempos de execução

O segundo algoritmo não-comparativo a ser testado, o Radix Sort apresentou excelentes resultados para todos os casos-teste, similares aos resultados apresentados pelo Counting Sort, mesmo quando os valores de k foram destoantes do valor de n.

5 Observações

Para a execução dos testes foi utilizado um computador com as seguintes configurações:

- Ubuntu MATE 16.04
- Unix Kernel 4.15.0-47
- GCC 5.4.0
- Go 1.12.1
- Processador i3-6006U
- 4GB RAM

De modo geral, identificamos que o Quick Sort apresentou o melhor tempo de execução de caso médio, o Merge Sort o melhor balanço dentre os modos de ordenação dos valores de entrada, o Insertion um bom funcionamento para valores ordenados/quase-ordenados e Radix/Counting excelentes resultados de tempo, embora com condições adicionais para bom funcionamento.