TDE 03 - Resolução de Problemas Estruturados em Computação

Gustavo Cesar Regnel

[1]Pontifícia Universidade Católica do Paraná (PUCPR)

Resumo. *Neste trabalho será demonstrado a implementação dos Métodos Insert Sort, Shell Sort e Merge Sort. Será exibido em formato de tabela os números de trocas, números de iterações e tempo de execução, como também a média desses 3 itens.*

1. Insert Sort:

Neste trabalho foi realizada a implementação do algoritmo de ordenação Insertion Sort, implementei o algoritmo de ordenação Insertion Sort e conduzi uma análise de desempenho em diferentes tamanhos de vetores aleatórios. Comecei criando um array chamado ”tamanhosVetor” que contém os tamanhos de vetor que quero avaliar. Em seguida, para cada tamanho especificado, gerei um vetor aleatório. Utilizei o algoritmo Insertion Sort para ordenar cada vetor aleatório e medi o tempo decorrido, o número de trocas e o número de iterações necessárias para a ordenação.

2. Shell Sort:

Também foi escolhida a implementação do algoritmo de ordenação Shell Sort em diferentes tamanhos de vetores aleatórios. Para cada tamanho especificado, o código gera um vetor aleatório e, em seguida, aplica o algoritmo Shell Sort para ordená-lo. Ele mede o tempo de execução, o número de trocas e o número de iterações necessárias para a ordenação. O Shell Sort é um algoritmo de ordenação que divide o vetor em segmentos pela metade, reduzindo gradualmente o espaço entre os elementos a serem comparados.

3. Merge sort:

Por final, foi escolhido também o Merge Sort para ser implementado. Inicialmente o vetor não ordenado será subdividido em dois subvetores de tamanhos próximos, esse processo é repetido recursivamente até que cada subvetor tenha um único elemento, o que é a condição de parada.

Primeiramente iremos analisar o número de trocas do método Insert Sort:

Tamanho:	50	500	1000	5000	10000
1 rodada:	669	60707	244487	6332278	24918354
2 rodada:	590	63128	243775	6312339	25205933
3 rodada:	600	62132	255035	6223899	25074910
4 rodada:	580	65190	253623	6352873	25056992
5 rodada:	604	61660	251514	6240781	24984536
média:	608,6	62.563,4	249.686,8	6.292.434	25.048.145

Table 1. Caption

Apesar de ser um método estável, percebe-se o aumento exponencial no número de trocas conforme aumentamos o tamanho de vetores. No final do processo, por estarmos lidando cada vez com vetores maiores, acabamos tendo números bem grandes.

Agora, vamos analisar o número de iterações:

Tamanho:	50	500	1000	5000	10000
1 rodada:	49	499	999	4999	9999

Table 2. Caption

O número de iterações em uma InsertSort deve ser sempre (n-1) em relação ao número total de vetores, pois é o total de comparações do vetor com o elemento anterior a ele. Ou seja, não faz sentido fazermos 5 rodadas visto que o número de iterações será sempre o mesmo.

E o tempo de execução(ms):

Tamanho:	50	500	1000	5000	10000
1 rodada:	0	9	2	7	32
2 rodada:	0	2	4	14	38
3 rodada:	0	3	1	10	27
4 rodada:	0	2	4	18	43
5 rodada:	0	3	2	10	27
média:	0	3,8	2,6	11,8	33,4

Table 3. Caption

Nessa análise, percebemos que entre 50/500 e 500/1000, o tempo de execução nos vetores menores é maior. Isso se justifica pela diferença proporcional, pois entre 50 e 500 há uma diferença de 10 vezes, e entre 500 e 100 apenas 2 vezes.

Agora vamos para o Shell Sort e seu número de trocas:

Tamanho:	50	500	1000	5000	10000
1 rodada:	139	2997	7663	61285	149954
2 rodada:	169	3390	7646	57865	161380
3 rodada:	200	2699	7622	62163	154046
4 rodada:	167	2765	7007	56194	150290
5 rodada:	136	3205	7274	61305	144236
média:	162,2	3.011,2	7.442,4	59.762,4	151.981,2

Table 4. Caption

Nessa análise, percebemos que o número de trocas aumenta à medida que o tamanho do vetor cresce, o que é esperado. O Shell Sort é um algoritmo de ordenação que envolve trocas de elementos para posicionar os elementos corretamente, e o número de trocas é diretamente proporcional ao tamanho do vetor.

Número de iterações:

Tamanho:	50	500	1000	5000	10000
1 rodada:	203	3506	8006	55005	120005

Table 5. Caption

Neste caso o número de iterações sempre será o mesmo, portanto, não existe necessidade de realizar 5 rodadas de teste. No código implementado, o vetor de entrada está sendo gerado de uma forma que é favorável ao Shell sort. O programa está gerando números aleatórios de 0 a 9999. Ou seja, com a entrada sendo sempre a mesma, o resultado será uniforme.

Tempo de execução(ms):

Tamanho:	50	500	1000	5000	10000
1 rodada:	0	1	1	4	3
2 rodada:	3	0	2	3	3
3 rodada:	0	1	0	2	3
4 rodada:	0	0	1	3	10
5 rodada:	0	1	1	2	1
média:	0,6	0,6	1	2,8	4

Table 6. Caption

O tempo de execução aumenta na medida em que o número de vetores aumenta, como o share sort é um algoritmo de ordenação que envolve comparações e movimentações de elementos, e o tempo de resposta geralmente aumenta com o tamanho do vetor.

Por fim, o Merge Sort e seu número de trocas:

Tamanho:	50	500	1000	5000	10000
1 rodada:	101	1906	4299	27068	59188
2 rodada:	105	1912	4352	27158	59138
3 rodada:	98	1898	4304	27033	59139
4 rodada:	104	1883	4295	27076	59099
5 rodada:	98	1895	4320	27044	59212
média:	101,2	1.898,8	4.314	27.075,8	59.155,2

Table 7. Caption

Nessa análise, percebemos que o Merge Sort é eficiente para ordenar vetores de diferentes tamanhos, com o número de trocas aumentando de acordo com o aumento do tamanho do vetor. Isso é consistente com o comportamento esperado do Merge Sort, que possui uma complexidade de tempo de O(n log n) e é amplamente utilizado para ordenação eficiente de conjuntos de dados de grande escala. Outro ponto interessante é a pequena diferença entre dos números com o mesmo vetor nas diferentes rodadas.

Seu número de iterações:

Tamanho:	50	500	1000	5000	10000
1 rodada:	286	4488	9976	61808	133616

Table 8. Caption

No código implementado, o número de iterações é incrementado no momento em que os elementos são copiados para os vetores da esquerda e da direita. Como essas cópias vão ocorrer em cada chamada recursiva, o número de iterações será sempre o mesmo para os vetores de mesmo tamanho.

Tempo em ms:

Tamanho:	50	500	1000	5000	10000
1 rodada:	0	1	1	3	7
2 rodada:	0	1	0	2	5
3 rodada:	0	0	0	2	4
4 rodada:	0	1	1	3	6
5 rodada:	0	1	0	3	5
Média:	0	0,8	0,4	2,6	5,4

Table 9. Caption

O tempo de execucão do código é relativamente baixo se comparado com os outros algoritmos, principalmente nos vetores mais baixos, o que demonstra que o Merge Sort é um algoritmo eficiente e com desempenho consistente.