Primeiro Projeto
SCC 0224/0606 – Estrutura de Dados II
Prof. Robson L. F. Cordeiro

João Victor de Almeida - 13695424
João Marcelo - 13835472
Daniel Umeda - 13676541

Introdução

Neste trabalho, implementamos alguns algoritmos de ordenação, para aplicarmos os conceitos aprendido em aula, com a finalidade de analisar sua complexidade e eficiência. Essa análise foi feita através do tempo de execução de cada algoritmo, observando quanto tempo leva para realizar a ordenação, de acordo com a quantidade “n” de elementos inseridos de forma aleatória na lista ligada, através de um TAD. Através de um gráfico, foi possível definir como seria sua complexidade, analisando de forma empírica e teórica. Para a análise de cada código utilizamos entradas “n” de 1000, 10000, 100000 ou mais elementos, fazendo 10 repetições cada, para que tenha uma maior precisão na análise da complexidade do algoritmo. Também, para cada entrada havendo 3 variações, que são, vetor aleatoriamente sortido, vetor aleatório já ordenado de forma crescente e ordenado de forma decrescente. Como o algoritmo auxiliar usado para ordenar as listas de forma crescente tem um tempo de execução muito grande, em alguns casos não foi possível verificar o tempo de execução do algoritmo de ordenação em questão.

Bubble-Sort:

O bubble-Sort é um algoritmo de ordenação de ordem quadrática em seu caso médio e pior caso, entretanto se a lista tiver ordenada, ele apresenta uma complexidade linear. Isso se dá pois o bubble compara dois a dois cada elemento do vetor e se o elemento anterior for maior que seu sucessor há troca.

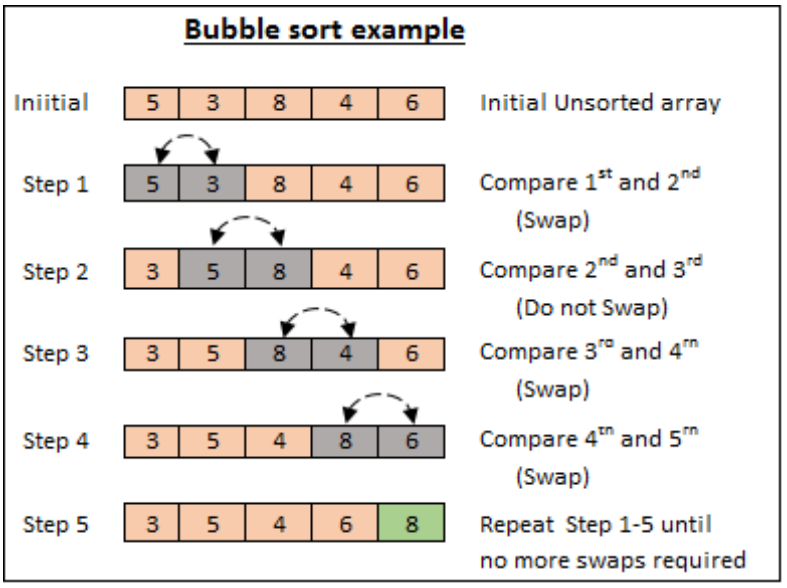

Foto Ilustrativa do Bubble.

Análise assintótica:

Bubble Sort: Ao realizar a análise, o total de operações realizadas pelo algoritmo é $(n^2 + 3n)/2$. Portanto, complexidade é $O(n^2)$ e $\Theta(n^2)$.

Bubble Sort Aprimorado: O total de operações realizadas pelo algoritmo é $n*(n+5)/2 + 3$. Portanto, a complexidade também é $O(n^2)$ e $\Theta(n^2)$.

Discussão sobre resultados:

Perante as entradas de “n” elementos tivemos os seguintes resultados, analisando pela tabela:

Vetor aleatoriamente gerado:

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00338	-2,42
10000	0,36910	-0,433
100000	43,105	1,634

Vetor ordenado(crescente):

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00105	-2,979
10000	0,18966	-0,722
100000	19,62116	1,293

Comparando o vetor aleatoriamente gerado, com o já ordenado, fica claro a diferença de tempo médio de execução, pois como citado acima o bubble-sort se torna com uma complexidade linear.

Quick-Sort:

O quick sort é um método de ordenação recursivo que se baseia nas seguintes etapas: escolher um pivô, dividir a lista em duas partes (uma com elementos maiores que o pivô e

outra com os menores), fazer isso recursivamente até que o número de elementos seja 1, combinar as partes ordenadas e formar a lista final.
A complexidade do quick sort depende do tamanho das partições e da escolha do pivô.

Análise assintótica:

A complexidade do Quick-Sort é O(n log n) no caso médio e Θ(n^2) no pior caso, que pode ser quando se escolhe o pior pivô possível (os das pontas das partições) e/ou quando o vetor já está ordenado)

Discussão sobre resultados:

Perante as entradas de “n” elementos tivemos os seguintes resultados, analisando pela tabela:

Vetor aleatoriamente gerado:

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0	-3,387
10000	0,00342	-2,466
100000	0,02312	-1,636
1000000	0,28213	-0,550
10000000	3,12667	0,495

Vetor ordenado(crescente):

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00901	-2,045
10000	0,35994	-0,444
100000	38,44443	1,585

Heap-Sort:

O heap-sort é um algoritmo de ordenação baseado na estrutura de dados heap. Tal estrutura é uma árvore binária onde cada nó pai é maior ou menor que seus filhos, a depender da ordem de ordenação desejada. A ordenação é feita tirando o maior elemento da árvore e o colocando no final da lista. É um algoritmo instável, visto que elementos iguais podem ser reposicionados.

Análise assintótica:

A complexidade do Heap-Sort é O(n log n)em qualquer caso, pois a etapa do Heapify tem complexidade O(logn), sendo n o número de elementos da lista. Como o heapify é rodado n vezes, a complexidade do algoritmo é O(n log n).

Discussão sobre resultados:

Vetor aleatoriamente gerado:

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00041	-3,387
10000	0,00341	-2,467
100000	0,03998	-1,398
1000000	0,58422	-0,233
10000000	8,82574	0,946

Vetor ordenado (crescente):

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00075	-3,125
10000	0,00263	-2,580
100000	0,03144	-1,506

Radix-Sort:

O radix sort é um algoritmo de ordenação estável que ordena seus elementos a partir da posição de cada dígito. O algoritmo divide a lista em “buckets” em que cada um deles representa um dígito, comparando primeiramente o menos significativo de cada elemento da lista. No vetor a seguir, por exemplo: [100, 099, 056, 102, 203], depois de cada iteração os buckets e o vetor ficariam assim:

Buckets da primeira iteração:
B(0): 100
B(1): __
B(2): 102
B(3): 203
B(4): __
B(5): __
B(6): 056
B(7): __
B(8): __
B(9): 099
[100, 102, 203, 056, 099]

Buckets da segunda iteração (2° dígito):
B(0): 100, 102, 203
B(1): __
B(2): __
B(3): __
B(4): __
B(5): 056
B(6): __
B(7): __
B(8): __
B(9): 099
[100, 102, 203, 056, 099]

Buckets da terceira iteração (1° dígito):
B(0): 056, 099
B(1): 100, 102, 203
B(2): __
B(3): __
B(4): __
B(5): __
B(6): __
B(7): __
B(8): __
B(9): __
[056, 099, 100, 102, 203] -> ordenado

Análise assintótica:

Entre as linhas 4 e 7 o loop é executado d vezes e counting sort é chamada uma vez dentro do loop, assim a complexidade é O(d*n) no pior caso, onde d é o número de dígitos no número mais longo. . A complexidade do counting sort é O(n), pois há 2 iterações independentes. A complexidade do Radix Sort é O(d*n*counting_sort), ou O(d*n²) no pior caso.

Discussão sobre resultados:

Com o algoritmo do radix sort em mãos, analisamos alguns casos de entradas para verificarmos o tempo de realização:

Vetor gerado aleatoriamente:

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0	-3,125
10000	0,00282	-2,550
100000	0,01583	-1,800
1000000	0,16499	-0,782
10000000	2,01191	0,304

Vetor ordenado (crescente):

Elementos(n)	Tempo Médio(s)	Log do tempo
1000	0,00085	-3,071
10000	0,00172	-2,765
100000	0,01604	-1,024

Conclusão:

De acordo com os resultados obtidos, vemos que o radix-sort foi o mais eficiente nos casos analisados, seguido pelo quick, que explode quando o vetor já está ordenado. O heap-sort,

ficou em 3° lugar, mas por ser um algoritmo de complexidade constante em qualquer caso, pode ser preferido dependendo das necessidades. Já o bubble-sort, foi claramente o mais lento de todos