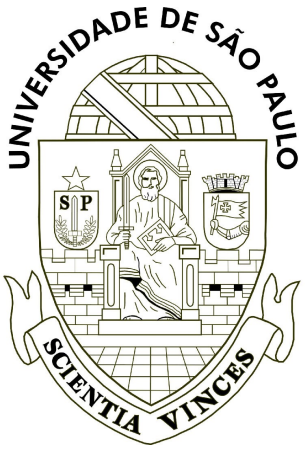
Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

SCC0606 - Estrutura de Dados II



Projeto 1

Alunos:
Lucas Lima Romero (13676325)
Luciano Gonçalves Lopes Filho (13676520)
Marco Antonio Gaspar Garcia (11833581)
Professor: Robson Leonardo Ferreira Cordeiro

Abril

2023

Sumário

1 Introdução

A ordenação é um problema fundamental em ciência da computação, sendo essencial em áreas como bancos de dados, redes, algoritmos de busca, entre outros. Compreendendo os diferentes algoritmos de ordenação e suas características, é possível escolher o algoritmo mais adequado para resolver um determinado problema.

Esse relatório está estruturado da seguinte forma:

Após a introdução, tem-se a seção que compila todas as análises assintóticas para os algoritmos de ordenação solicitados. Essa parte é seguida da análise empírica, com as tabelas e os gráficos associados aos tempos de execução de cada algoritmo. Finalmente, as conclusões foram apresentadas seguidas da bibliografia, base teórica das informações exprimidas nesse relatório.

O intuito deste trabalho é analisar quatro algoritmos diferentes, tanto em termos teóricos quanto empíricos, e discutir seus melhores e piores casos.

Sobre o trabalho, a priori todo o código foi elaborado pelos participantes, com base nas aulas e nos pseudocódigos fornecidos pelo professor. Os comentários do próprio código apresentam explicações detalhadas sobre a estrutura e os algoritmos. Foram testadas todas as funções de ordenação, até chegar ao código com as saídas enxutas, que se resumem

apenas aos tempos de execução e os desvios padrões deles. A partir disso, todo o relatório foi feito, se baseando nos resultados obtidos.

2 Análise Assintótica

2.1 Bubble Sort e Bubble Sort Aprimorado

Com o código utilizado para o Bubble Sort sem otimizações, a análise teórica pode ser contabilizada a partir dos laços aninhados. Tem-se uma condicional que realiza a troca, com preço constante 5 (associados as operações de atribuições e somas), que por sua vez são executadas $(n-1)^2$ vezes.

Tem-se, portanto, que o preço final é de aproximadamente: $5n^2 - 10n + 5$. Logo o algoritmo é quadrático, isto é, da ordem de $O(n^2)$.

Já o Bubble aprimorado conta com uma variável que checa se houve trocas naquela interação, além de contar com um número cada vez menor de comparações, já que a cada rodada tem-se a certeza de que o último elemento está no lugar certo.

Na implementação feita pelo grupo, o preço constante dentro dos laços aninhados passa a ser 6, as quais, no pior caso são executadas $\frac{n^2}{2}$ vezes.

Logo, tem-se que o preço final é de $3n^2$, com a mesma ordem de $O(n^2)$.

2.2 Quick Sort

Na implementação do grupo, tem-se o preço constante de chamada inicial da recursão, igual a 2 (chamada e subtração). A cada chamada recursiva tem-se um preço da metade da entrada anterior, além de um preço constante de aproximadamente 8. Isso gera uma árvore de recursão de forma que o custo final é proporcional a $\log n$ (altura da árvore) vezes n, custo por nó.

Logo, tem-se que o preço desse algoritmo é de aproximadamente $8n \log n + 2$, sendo da ordem de $\Theta(n \log n)$. Com a escolha do pivô sendo sempre o menor ou o maior valor a cada chamada recursiva, a altura da árvore deixa de ser $\log n$ e passa a ser n, de modo que esse algoritmo recai ao seu pior caso, em que é quadrático da ordem de $O(n^2)$.

2.3 Radix Sort

Esse método de ordenação é estruturado na análise dos dígitos dos menos aos mais significativos, de modo que a ordenação é feita a partir deles.

No algoritmo utilizado, tem-se um preço constante da função iterativa igual a $14n + 17$. Essa função é chamada pela principal, que, por sua vez, possui um custo fixo de $n - 1 + 6(14n + 17)$ resultando, no total em $85n + 101$. Logo, o algoritmo é da ordem de $O(n)$.

2.4 Heap Sort

A Complexidade da função heapify pode ser calculada considerando três fatores: a altura, o número de nós e o tempo de comparação entre valores (constante). É conhecido que a altura de uma árvore binária é $\log n$ e o número de elementos em um nível é $[n/2^{n+1}]$, sendo n a quantidade de elementos total da árvore.

$$\sum_{i=0}^{logn} \frac{n}{2^{n+1}} \times k$$

Resolvendo o somatório acima, chegamos a uma função da ordem $O(n)$. A Complexidade do procedimento heapsort é dependente da altura da árvore ($\log n$), e como esse procedimento chama o procedimento heapify, a complexidade final deve ser o produto entre a altura da árvore e a complexidade do procedimento heapify. Logo:

$$T(n) = n \log n = O(n \log n)$$

3 Análise Empírica

Mediu-se o tempo de execução de cada algoritmo de ordenação dez vezes para três tipos de vetores diferentes: aleatório, ordenado e decrescente. Cada um dos algoritmos se comporta de forma diferente conforme tipo do vetor utilizado. A média dos tempos de execução foram organizadas em tabelas, bem como os desvios padrões, para uma melhor visualização dos dados.

Após a análise das médias de tempo, foram plotados gráficos em Python, que representam os tempos de execução de cada um dos algoritmos em função do tamanho de entrada. Foram utilizados valores de entrada intermediários, que não estão na tabela, para que fosse possível visualizar com mais clareza a tendência dos algoritmos.

3.1 Bubble Sort

Como visto anteriormente, o Bubble Sort é um algoritmo de complexidade quadrática, e por isso os tempos de execução escalam rapidamente. Seria inviável medir o tempo para uma ordem de grandeza maior do que 10^5 elementos.

Tabela 1: Tempos de execução do Bubble Sort

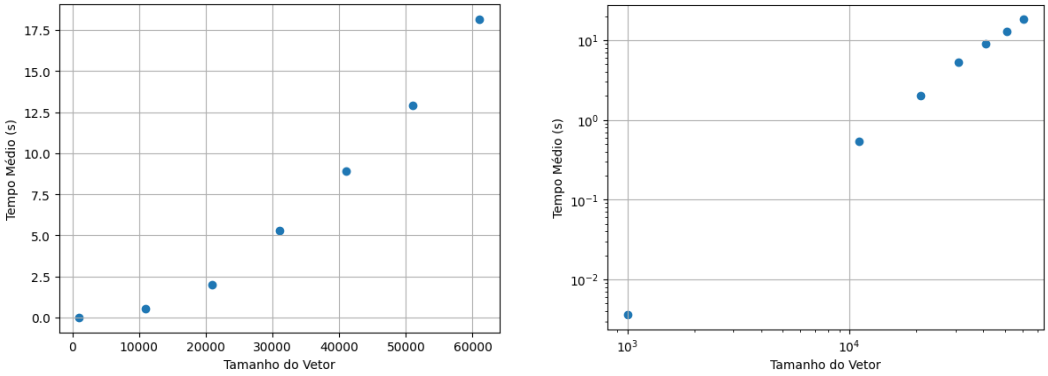
Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0048880000	0.0013021999	0.0021778001
10^4	0.2352955043	0.1298687011	0.2159539014
10^5	28.5255393982	13.3947019577	22.0985736847

Tabela 2: Desvio Padrão do Bubble Sort

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0008560820	0.0000155592	0.0000336245
10^4	0.0032163768	0.0009396118	0.0004312723
10^5	0.1106043554	0.0307460281	0.0318697546

No Bubble Sort, o pior caso mostrou-se no vetor de números aleatórios, e o melhor caso quando o vetor já está ordenado.

Figura 1: Bubble Sort



3.2 Bubble Sort Aprimorado

Os tempos de execução da versão aprimorada foram consideravelmente menores do que a versão original, entretanto, continuam escalando rapidamente.

Tabela 3: Tempos de Execução do Bubble Sort Aprimorado

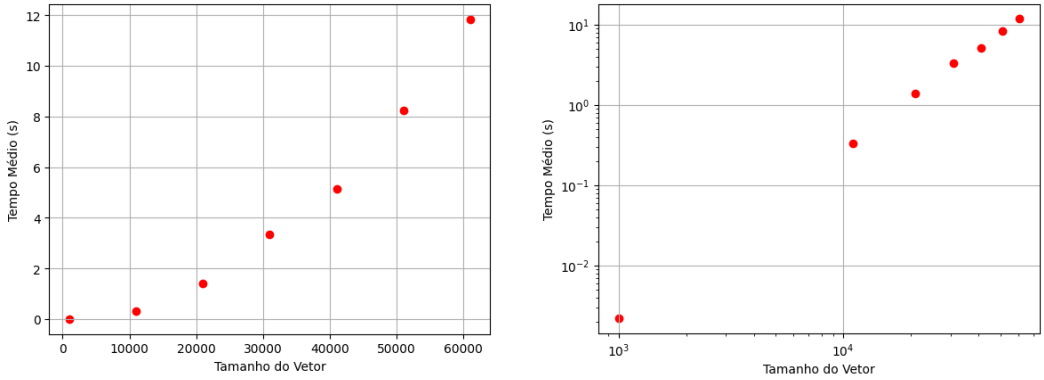
Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000056400	0.0000019000	0.0016381999
10^4	0.0028912418	0.0000144000	0.1609017998
10^5	22.5040664673	0.0001448000	16.4473381042

Uma grande diferença desta versão para a outra é a verificação de mudança, por isso, no vetor ordenado, o algoritmo percorre o vetor apenas uma vez e termina a execução. O pior caso continuou sendo o do vetor de aleatórios.

Tabela 4: Desvio Padrão do Bubble Sort Aprimorado

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000056400	0.0000004583	0.0000136195
10^4	0.0028912418	0.0000007000	0.0005118433
10^5	0.0585869096	0.0000067409	0.0109472515

Figura 2: Bubble Sort Aprimorado



3.3 Quick Sort

Curiosamente, o pior caso do Quick Sort ocorreu quando o vetor já estava ordenado. No caso decrescente, também percebe-se que o tempo de execução é extremamente maior do que o caso aleatório.

Tabela 5: Tempos de Execução do Quick Sort

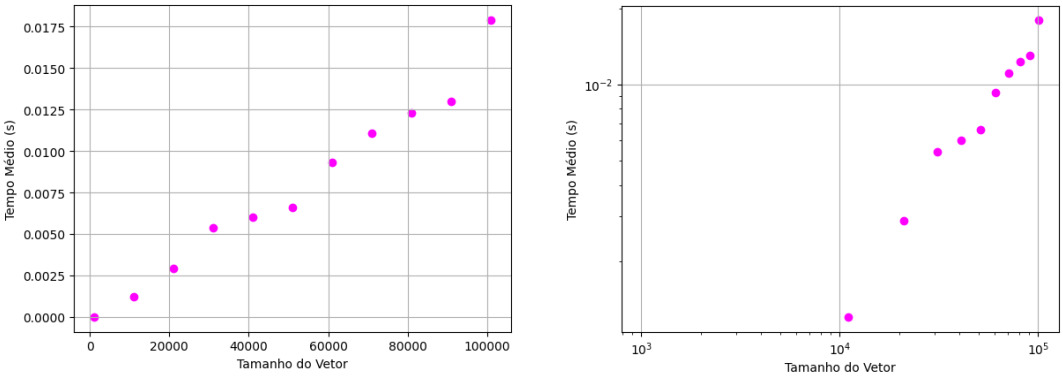
Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000490000	0.0011849001	0.0009224000
10^4	0.0007326000	0.1156996042	0.0893282965
10^5	0.0093319006	11.8322381973	9.0932960510

Os dados da tabela indicam que o Quick Sort não é recomendado para listas quase ordenadas, uma vez que o algoritmo não verifica inicialmente se a lista já está ordenada, portanto, embaralhando-a para ordená-la novamente.

Tabela 6: Desvio Padrão do Quick Sort

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000046433	0.0000060033	0.0000182398
10^4	0.0000189008	0.0002921534	0.0003311032
10^5	0.0000845852	0.0083834362	0.0083593241

Figura 3: Quick Sort



3.4 Radix Sort

O algoritmo que melhor performou dentre todos os analisados foi o Radix Sort. Devido à sua eficiência, decidiu-se analisá-lo em maiores ordens de grandeza, adicionando entradas de um milhão de elementos até cem milhões, e o melhor caso apresentou-se no vetor de números aleatórios.

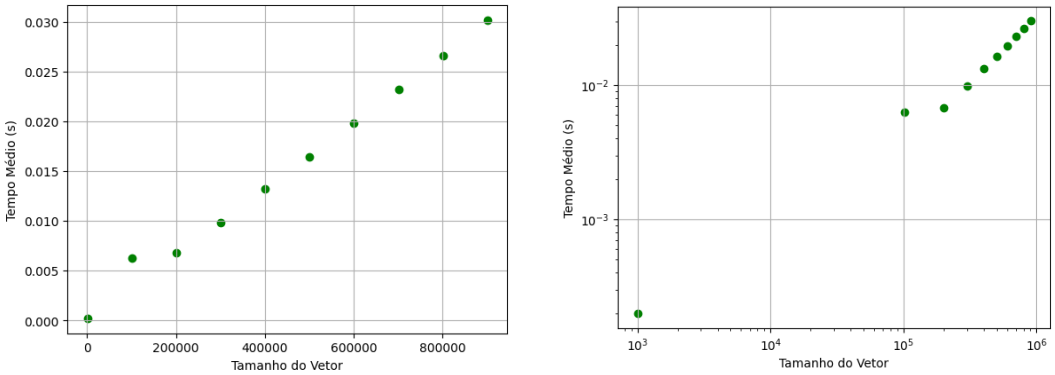
Tabela 7: Tempos de Execução do Radix Sort

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000309000	0.0000254000	0.0000249000
10^4	0.0003130000	0.0003180000	0.0003099000
10^5	0.0032953999	0.0039849998	0.0039325999
10^6	0.0340274982	0.0472413003	0.0467645004
10^7	0.3493627000	0.5531321000	0.5652028000
10^8	3.8499754000	6.8292770000	7.1561989000

Tabela 8: Desvio Padrão do Radix Sort

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000015620	0.0000007000	0.0000006633
10^4	0.0000084528	0.0000085299	0.0000026476
10^5	0.0000549517	0.0000701776	0.0000417799
10^6	0.0007554170	0.0009207695	0.0014085131
10^7	0.0034580476	0.0061003934	0.0028106070
10^8	0.1761862567	0.1522321945	0.6451118030

Figura 4: Radix Sort



3.5 Heap Sort

O Heap Sort também foi extremamente veloz, se comparado aos outros algoritmos, por isso, adicionou-se mais ordens de grandeza na análise.

Tabela 9: Tempos de Execução do Heap Sort

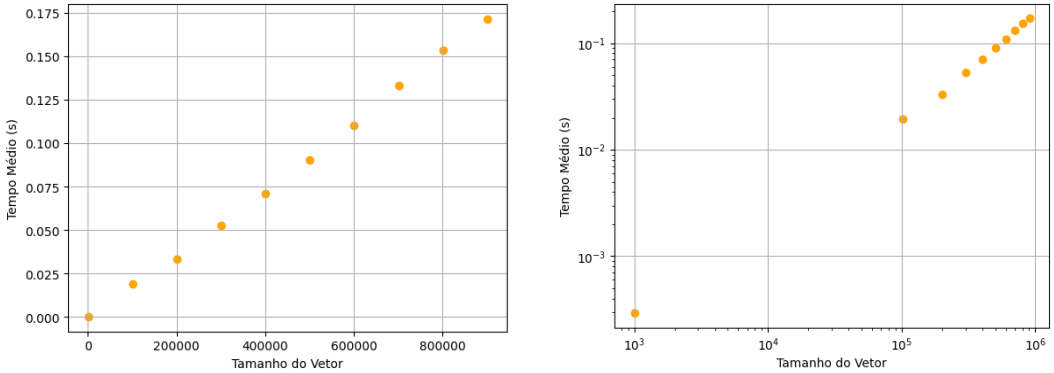
Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0000852000	0.0000792000	0.0000725000
10^4	0.0012269000	0.0009776000	0.0009573000
10^5	0.0158641003	0.0113629997	0.0110472003
10^6	0.1979769021	0.1304377913	0.1287351996
10^7	3.3283270000	1.4766264000	1.5053195000
10^8	47.1186552000	16.5191334000	16.8931802000

Dessa vez, o pior caso foi o do vetor de números aleatórios, e apresentou um equilíbrio entre os casos ordenado e decrescente. Entretanto, o Radix Sort ainda mostrou-se muito mais veloz que o Heap Sort, comprovando a complexidade obtida da análise assintótica.

Tabela 10: Desvio Padrão do Heap Sort

Tamanho do Vetor	Aleatório (s)	Ordenado (s)	Decrescente (s)
10^3	0.0003000000	0.0004000000	0.0003000000
10^4	0.0004000000	0.0004582576	0.0004898979
10^5	0.0086602540	0.0095582425	0.0011000000
10^6	0.0255960935	0.0032310989	0.0156425701
10^7	0.0707530976	0.0073345308	0.0025876720
10^8	0.7041794909	0.0523709405	0.1688951714

Figura 5: Heap Sort



4 Conclusões

Após a análise dos algoritmos, concluiu-se que, apesar de muito mais intuitivos, os métodos de ordenação quadráticos são inviáveis em aplicações reais. Sem levar em conta as complexidades de espaço, os algoritmos $O(n)$ e $O(n \log n)$ são extremamente eficientes na ordenação de listas, como pôde ser observado no Radix Sort e Heap Sort.

Uma das dificuldades enfrentadas pelo grupo foi a análise assintótica dos algoritmos, haja vista que, além da análise Big-oh, foi necessária uma análise preliminar mais rigorosa de cada algoritmo. Dessa maneira, pode-se dizer que isso foi mais complexo que a análise empírica, pois exigiu dos integrantes a contagem dos tempos de execução de cada linha, de forma minuciosa e cadenciada.

Outra dificuldade foi a conciliação dos códigos dos estudantes. Para contornar esse problema, utilizamos a tecnologia Git, por meio do Github, para ter um repositório desse trabalho, de modo que todos os integrantes fossem colaboradores ativos, permitindo que todos trabalhassem sobre o mesmo código durante todo o processo de estruturação.

Bibliografia

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press & McGraw-Hill, 2001.

Tabelas: elaboradas pelos compiladores.