UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO - ICMC

Curso Engenharia de Computação

Projeto Prático I

Gustavo Curado Ribeiro 14576732

Lucien Rodrigues Franzen 14554835

Luís Filipe Silva Forti 14592348

São Carlos - SP

2024

1. Introdução:

Para este projeto, foi criado um código de ordenação de valores simples e ineficiente, buscando apresentar a complexidade de forma mais objetiva e perceptível. Após medidos os tempos de processamento, foi comparado os resultados teóricos com os empíricos. Para a segunda parte, a complexidade de 4 algoritmos desconhecidos foi analisada com base em seu tempo de processamento com diferentes tipos e tamanhos de entradas por meio da análise de seus gráficos de tempo em função do tamanho da entrada.

2. Código implementado:

```
#Luís Filipe Silva Forti 14592348
#Lucien Rodrigues Franzen 14554835
#Gustavo Curado Ribeiro 14576732

import time

#Elimiar o primeiro número, pois não é necessário
entrada = input()
#Ler a sequência numérica
entrada = input()

#Separa a sequência e salva
valores = [int(valor) for valor in entrada.split() if valor.isdigit()]

#Para passar por todas as casas
for i in range(0, len(valores)):
    #Para verificar todas as casas não-organizadas
    for j in range(i+1, len(valores)):
        #Se o valor da casa j for menor que o valor da casa i
        if valores[i] > valores[j]:
            #Salva o valor da casa i
            aux = valores[i]
            #Substitui pelo valor da casa j
            valores[i] = valores[j]
            #Salva o valor da casa i em j
            valores[j] = aux
            #Dessa forma, sempre que aparecer um valor menor que o valor em i,
            #ele irá trocar de lugar, organizando em forma crescente

#Para imprimir os valores separados por espaço
for valor in valores:
    print(valor, end = ' ')
```

O código é uma versão menos otimizada do método BubbleSort, em que este irá percorrer todo o processo independentemente da organização inicial dos valores. Ele segue o mesmo processo do BubbleSort, onde ele percorrerá casa por casa, comparando com o valor da casa sendo analisada e trocando sempre que necessário, mas, diferentemente do BubbleSort, ele não ficará verificando se já está organizado, encerrando antecipadamente, mas irá fazer todos os ciclos até o final.

Para a análise teórica de complexidade, foi iniciada pelo pior caso, onde os valores estão organizados de forma decrescente, forçando o código a realizar o máximo de trocas possível por ciclo. Para essa análise, a estrutura do código foi avaliada, levando em conta cada passo do processo, formulando funções matemáticas para cada etapa. Então, juntamos as funções de cada parte do código para conseguir a fórmula que descreve o código inteiro nesse caso.

O primeiro ciclo ocorrerá em sua íntegra, servindo de limites para o segundo ciclo. O segundo ciclo, por sua vez, ocorrerá um número de vezes dependente do primeiro ciclo, por conta do trecho “in range(**i**+1, …”, assim cria-se uma dependência entre os dois ciclos. Por conta de estar sendo analisado o pior caso, todo o código dentro do segundo ciclo ocorrerá

em cada passo, assim dependendo único e exclusivamente do segundo loop. Assim, define-se que todas as etapas do segundo ciclo ocorrerão uma quantidade de vezes dependendo da situação do primeiro ciclo, o que é caracterizado por um somatório (Σ) dependente do valor do primeiro ciclo. Por fim, também é necessário considerar a quantidade de operações do primeiro ciclo, o qual é independente do restante do código. Desta forma, obtém-se a função: $n + 1 + \sum_{i=0}^{n-2}((n - (i + 1)) * 5)$, onde a multiplicação por 5 é pelos processos dentro do segundo ciclo, sendo o "if" implícito do "for", o "if" explícito e as outras 3 alocações que ocorrem em seguida. O "n" é devido ao "if" implícito do primeiro "for" e a soma de 1 é por conta do "if" implícito do segundo "for", o qual ocorre no último ciclo do processo, onde ele imediatamente sai. Esta função pode ser expandida, obtendo-se: $\frac{5}{2}n^2 - \frac{3}{2}n + 1$.

Para o melhor caso, onde os valores estão organizados de forma crescente, nenhuma troca ocorre na ordenação, ou seja, todos os passos ocorrem, exceto os de substituição dos valores. Devido à ineficiência do código, todos os ciclos ocorrem em sua totalidade, obtendo-se assim a função: $n + 1 + \sum_{i=0}^{n-2}((n - (i + 1)) * 2)$. A troca da multiplicação de 5 por 2 é devido à falta da etapa de troca dos valores. Esta função pode ser expandida, obtendo-se: $n^2 + 1$.

Assim, dadas as funções obtidas, pode-se afirmar que a complexidade do algoritmo é $\Omega(n^2)$ para o melhor caso e $O(n^2)$ para o pior. Como as funções de ambos os casos possuem complexidades assintóticas equivalentes, conclui-se que o comportamento médio do algoritmo é $\Theta(n^2)$, o que também é perceptível nas tabelas abaixo, pois a taxa de crescimento da proporção do tempo de um caso em relação ao seu anterior é quadrática.

Ordenados de forma crescente		
Nº de entradas	Tempo (s)	Proporção em relação ao anterior
100	0,0003527	———
500	0,0083578	23,69
1000	0,0366075	4,38
5000	0,8790637	24,01
10000	3,5405134	4,03

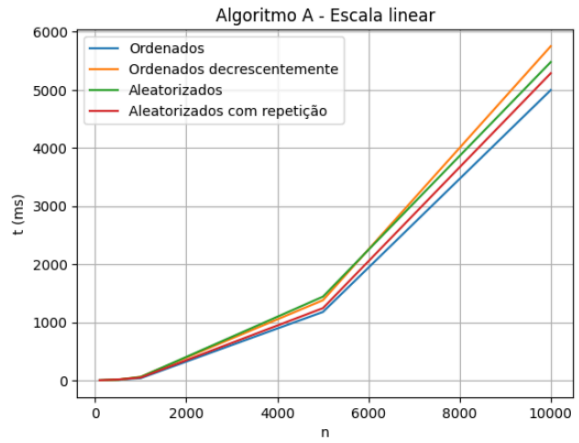
Ordenados de forma decrescente		
Nº de entradas	Tempo (s)	Proporção em relação ao anterior
100	0,0008506	———
500	0,0202450	23,80
1000	0,0836504	4,13
5000	2,0759794	24,82
10000	8,9481129	4,31

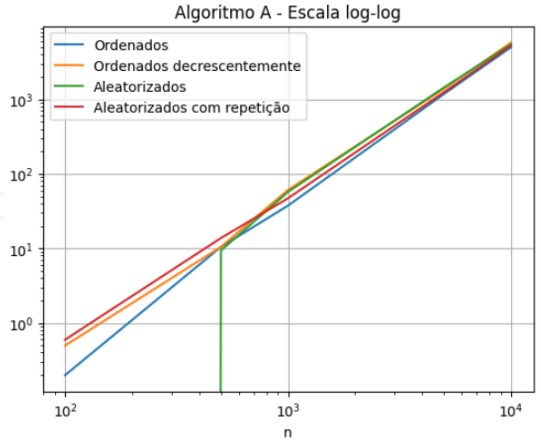
Aleatorizados		
Nº de entradas	Tempo (s)	Proporção em relação ao anterior
100	0,0004999	———
500	0,0146469	29,30
1000	0,0586869	4,01
5000	1,4535851	24,77
10000	5,9046886	4,06

Aleatorizados com repetição		
Nº de entradas	Tempo (s)	Proporção em relação ao anterior
100	0,0005000	———
500	0,0128519	25,70
1000	0,0545747	4,25
5000	1,3194789	24,18
10000	5,4935188	4,16

3. Baseline de algoritmos:

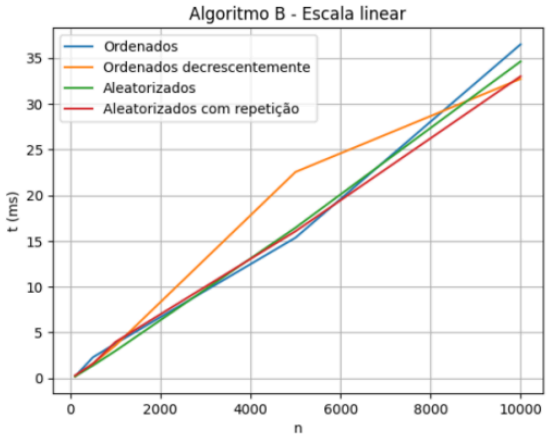
a. Algoritmo A:

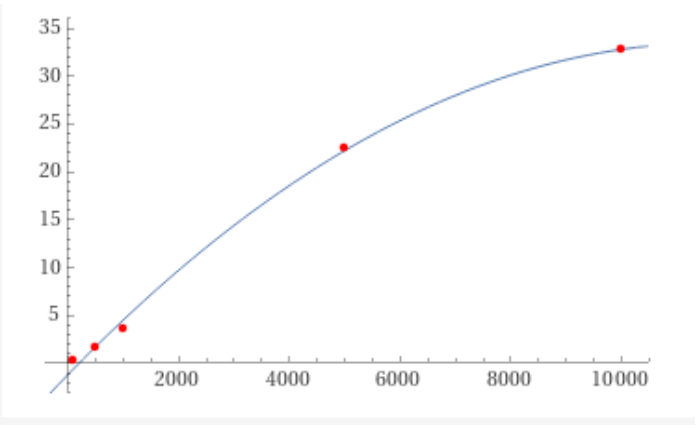


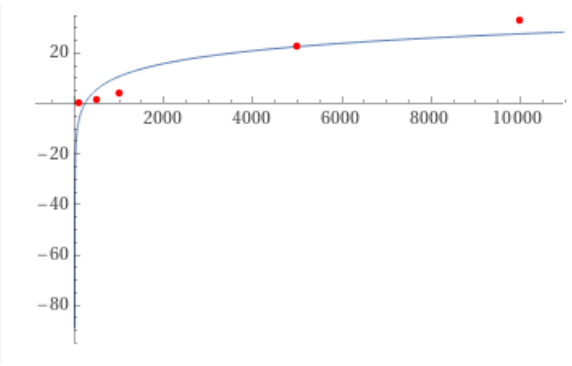


Pode-se observar que todos os casos apresentam curva de crescimento similar. Além disso, o plot dos valores na escala log-log linearizou o crescimento dos valores, o que é característico de curvas polinomiais. Calculando-se o coeficiente angular da reta na escala log-log para o caso ordenado, obtém-se aproximadamente 2,2. Portanto, conclui-se que o algoritmo possui complexidade $\Theta(n^2)$.

b. Algoritmo B:

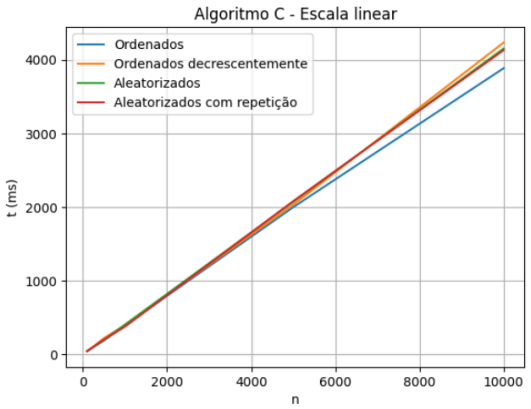






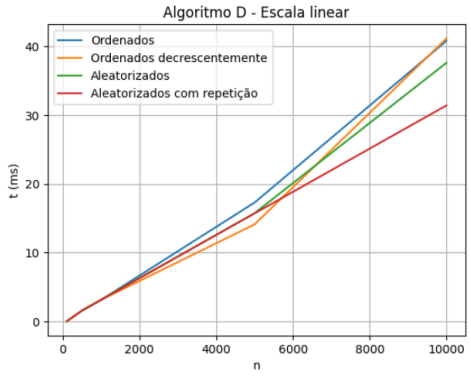
Por mais que as curvas dos casos aleatorizados sejam próximas a uma reta, as dos casos ordenados são mais inconsistente. Caso tente-se plotar esses casos como curvas quadráticas, acaba que o caso ordenado decrescentemente se torna uma parábola invertida (2ª imagem), o que não é válido. Além disso, a tentativa de plotar o caso ordenado decrescentemente como uma curva logarítmica não gerou bons resultados (3ª imagem). Portanto, é mais provável que esse algoritmo seja Θ(n), ou que faltem dados para que se possa visualizar a curva real da função tempo.

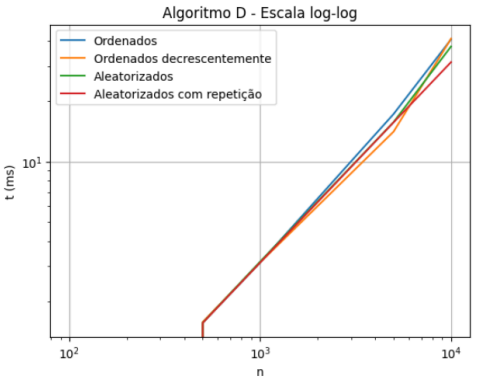
c. Algoritmo C:



É evidente pelo gráfico que a curva de crescimento é linear para todos os casos desse algoritmo. Portanto, é seguro afirmar que a complexidade média do algoritmo é Θ(n).

d. Algoritmo D:





Nesse algoritmo, os casos aleatorizados parecem ter crescimento linear e os ordenados aparentam, à primeira vista, ter crescimento diferente. No entanto, as tentativas de calcular o coeficiente angular dos casos ordenados na escala log-log para determinar o grau do polinômio geram resultados próximos a 1, o que indicaria ou uma reta, ou uma parábola com coeficiente ”a” muito pequeno. Além da forma visual da curva dos ordenados corroborar mais para a segunda hipótese, tentativas de aproximar esses casos para retas e funções quadráticas demonstraram que a curva quadrática se aproxima mais do resultado obtido. Assim, é provável que o algoritmo possa ser classificado como Ω(n), sendo o seu melhor caso o de entrada aleatorizada com repetições, e $O(n^2)$, sendo o pior caso o de entrada ordenada decrescentemente.

4. Conclusão:

Neste trabalho, foram avaliados em complexidade alguns algoritmos de ordenação, utilizando seus tempos de execução como base. Uma das dificuldades do trabalho foi a necessidade de definir um somatório, assim como seus limites, na fórmula de complexidade do algoritmo escolhido para a primeira parte. Em relação à segunda parte, o algoritmo que trouxe maior dificuldade foi o algoritmo B, dados os seus valores inconsistentes quando os dados de entrada eram ordenados de forma decrescente, destoando consideravelmente dos demais casos.

Quando compara-se o algoritmo implementado com os algoritmos fornecidos, percebe-se que o algoritmo A é o mais próximo do implementado, pois ambos possuem complexidade média $\Theta(n^2)$, sendo pouco eficientes quando comparados com os demais. O algoritmo mais eficiente para valores de entrada altos, por sua vez, é o algoritmo B, pois é o que apresentou os menores valores de tempo e, junto com o C, possui a menor complexidade dentre os algoritmos. O algoritmo D, no entanto, se apresentou melhor para valores pequenos, mesmo que, para os casos ordenados, esse algoritmo possui complexidade $O(n^2)$, o que deve afetar sua eficiência para entradas maiores que 10000.

Por fim, percebe-se que o algoritmo C, por mais que tenha a menor complexidade, mostrou-se extremamente ineficiente para os valores fornecidos, sendo apenas mais rápido que o algoritmo A, seja em casos com poucas ou muitas entradas e independentemente da organização original, e melhor que o C para valores muito maiores organizados de forma decrescente.