Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

Análise e comparação de eficiência e custo entre algoritmos de ordenação.

Stefany Vitória da Conceição Izidio.

Recife

Setembro de 2020

Resumo

Visando a importância da ordenação de dados na eficiência de tarefas como a de busca, esta pesquisa tem o objetivo de avaliar por meio de experimentos e comparações os algoritmos de ordenação bubble sort, insertion sort e quick sort. E apontar em quais situações um algoritmo tem melhor performance sobre o outro. Para tanto, são executados testes em 3 bases de dados que possuem características diferentes, em cada algoritmo e, os dados são catalogados, analisados e comparados. Assim tornando possível chegar a conclusão de qual algoritmo tem melhor desempenho e em quais situações convém optar pelo uso deles.

Palavras-chave: Ordenação, bubble sort, insertion sort, quick sort, algoritmos, desempenho.

Abstract

Aiming at the importance of ordering data in the efficiency of tasks such as search, this research aims to evaluate, through experiments and comparisons, the bubble sort, insertion sort and quick sort ordering algorithms. And point out in which situations one algorithm performs better than the other. For that, tests are performed in 3 databases that have different characteristics, in each algorithm and the data are cataloged, analyzed and compared. Thus making it possible to reach the conclusion of which algorithm has better performance and in which situations it is convenient to choose to use them.

Keywords: Sorting, bubble sort, insertion sort, quick sort, algorithms, performance.

1. Introdução

1.1 Apresentação e Motivação

Este trabalho é sobre algoritmos de ordenação, tais algoritmos são importantes pois visam facilitar a recuperação posterior de itens em um conjunto de dados. Exemplos que mostram a importância da ordenação são: a ineficiência em buscar um dado em um conjunto de dados grande e desordenado, como a dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética ou a dificuldade de encontrar uma determinada palavra em um dicionário desordenado. Diante dessas situações os algoritmos de ordenação têm por finalidade automatizar o processo de ordenamento e posteriormente tornar o processo de busca mais eficiente. O atual trabalho trata mais especificamente dos algoritmos denominados Bubble Sort, Insertion sort e Quick sort. No trabalho será feita uma pesquisa, análise e comparação entre os algoritmos supracitados, de modo a apontar qual o algoritmo mais eficiente e indicado em algumas situações, tal pesquisa, análise e comparação faz-se necessária, pois, este tipo de algoritmo é comumente usados em diversos projetos na área da computação, assim como em outras áreas e é de fundamental importância que a escolha tomada em relação ao qual algoritmo usar seja a correta para garantir a eficiência e garantir o menor custo possível nos projetos onde forem aplicados, pois, a escolha não exata do algoritmo de ordenação, pode causar ineficiência no processo de ordenamento, o que pode resultar no aumento do custo de tempo e/ou armazenamento. O trabalho tem como pergunta de pesquisa: Qual o algoritmo de ordenação mais eficiente em relação aos custos computacionais, como custo de processamento e complexidade, e em quais circunstância deve ser considerado o uso de tal algoritmo?

1.2 Objetivos

O objetivo geral do trabalho é apontar o/os algoritmo(os) mais eficiente(es) considerando o custo e o contexto em que será aplicado. O mesmo tem como objetivos específicos:

- Realizar pesquisas sobre os algoritmos e identificar os pontos fortes e fracos de cada um.
- Analisar o custo da ordenação e a complexidade dos algoritmos escolhidos.
- Desenvolver 3 bases de dados com características que interferem na performance dos algoritmos e simulam situações reais.

- Aplicar os algoritmos nas bases de dados e catalogar a performance de cada algoritmo.
- Comparar o desempenho de cada algoritmo quando aplicados sobre as mesmas bases de dados.

1.3 Organização do trabalho

A seção 1 contém a introdução do trabalho, a motivação, os objetivos e a organização do mesmo. A seção 2 contém o referencial teórico, onde o tema é explanado. A seção 3 contém o procedimento e método usado no trabalho, onde é discutido o material usado, as ferramentas e como o trabalho foi executado. Na seção 4 encontra-se os resultados do procedimento. Na seção 5 encontra-se as conclusões atingidas com a pesquisa.

2. Referencial Teórico

2.1 Algoritmos de ordenação

O atual trabalho tem o papel de explanar sobre os principais e mais populares algoritmos de ordenação. Esses algoritmos tem por finalidade organizar elementos de uma dada sequência. Existem diversos algoritmos baseados em diversas estratégias, no trabalho será feito a análise dos algoritmos Bubble sort, Insertion sort e Quick sort.

Analisar um algoritmo significa prever os recursos de que o algoritmo necessita. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, porém mais frequentemente é o tempo de computação que desejamos medir (THOMAS H. CORMEN, 2012).

Os algoritmos de ordenação são classificados em algoritmos de ordenação interna, que se dá quando a ordenação é feita na memória principal do computador, e algoritmos de ordenação externa, que ocorre quando a ordenação não cabe na memória principal do computador. Neste trabalho, serão utilizados somente métodos de ordenação interna. Durante a escolha de um algoritmo de ordenação o aspecto mais importante a se considerar é o tempo gasto durante a sua execução. Para algoritmos de ordenação interna, as medidas de custos mais relevantes são comparações entre chaves e o número de trocas de itens. A quantidade extra de memória auxiliar utilizada pelo algoritmo, como vaiáveis auxiliares, é também um aspecto que também deve ser considerado, pois o uso econômico da memória disponível é um requisito importante na ordenação interna.

Os métodos de ordenação interna também são classificados em métodos simples e métodos eficientes, geralmente os simples são indicados para pequenos arquivos, devido a fácil compreensão e implementação, enquanto os métodos eficientes são indicados para arquivos maiores. Os métodos simples tem um maior número de comparações em relação aos métodos eficientes.

Um método de ordenação pode ser estável ou instável, se a ordem relativa dos itens com chaves iguais mantém-se inalterada pós processo de ordenação, o algoritmo é dito estável, caso contrário é dito instável. Um exemplo de algoritmo estável é quando uma lista de alunos organizada em ordem alfabética é ordenada pelo fator nota e alunos com notas iguais aparecem na lista em ordem alfabética.

2.2 Bubble sort

O escritor CORMEN, Thomas H. et al. No seu grande livro Algoritmos: teoria e prática, diz: “O bubble sort é um algoritmo de ordenação popular, porém ineficiente. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem”. O bubble sort é um algoritmo de ordenação classificado como simples, estável e bastante popular no meio computacional. Também conhecido como algoritmo de ordenação por flutuação, este método de ordenação faz várias passagens por um vetor. Nas passagens compara itens adjacentes e faz a troca daqueles que estão fora da ordem determinada.

Pseudo-código do bubble sort.

```
BUBBLESORT(vetor):
    n = comprimento(vetor)
    para j=n-1, até 0, j--:
        para i=0, até j, i++:
            se vetor[i] > vetor[i+1]
                trocar vetor[i] <-> vetor[j]
```

Cada passagem pelo vetor coloca ao menos um elemento do vetor em seu devido lugar. Em alusão, cada item “borbulha” até o local ao qual pertence, essa movimentação lembra a forma como as bolhas flutuam em um tanque de água e procuram seu próprio nível, e disso vem o nome do algoritmo.

O bubble sort obtém seu pior desempenho quando a sequência se encontra inversamente ordenada. Devemos observar que independentemente de como os itens são previamente ordenados na entrada, serão feitas (n-1) passagens para ordenar uma sequência de tamanho n. No pior caso, cada comparação causará uma troca. O melhor caso ocorre

quando todos os elementos do vetor já estão ordenados, pois, se a sequência já estiver ordenada, nenhuma troca será feita. Em média, trocamos metade do tempo. Para qualquer caso a complexidade do algoritmo é O(n2). No anexo 1, encontra-se a análise de custo mais detalhada.

execução do bubble sort em ordenação crescente em um vetor de tamanho 8.

								Trocas
44	55	12	42	94	18	06	67	
44	55	12	42	94	18	06	67	55↔12
44	12	55	42	94	18	06	67	55↔42
44	12	42	55	94	18	06	67	
44	12	42	55	94	18	06	67	94↔18
44	12	42	55	18	94	06	67	94↔06
44	12	42	55	18	06	94	67	94↔67
44	12	42	55	18	06	67	94	44↔12
12	44	42	55	18	06	67	94	44↔42
12	42	44	55	18	06	67	94	
12	42	44	55	18	06	67	94	55↔18
12	42	44	18	55	06	67	94	55↔06
12	42	44	18	06	55	67	94	
12	42	44	18	06	55	67	94	
12	42	44	18	06	55	67	94	
12	42	44	18	06	55	67	94	44↔18
12	42	18	44	06	55	67	94	44↔06
12	42	18	06	44	55	67	94	
12	42	18	06	44	55	67	94	
12	42	18	06	44	55	67	94	42↔18
12	18	42	06	44	55	67	94	42↔06
12	18	06	42	44	55	67	94	
12	18	06	42	44	55	67	94	
12	18	06	42	44	55	67	94	18↔06
12	06	18	42	44	55	67	94	
12	06	18	42	44	55	67	94	12↔06
06	12	18	42	44	55	67	94	

Figura 1. Representação de execução do bubble sort.

Fonte: Glauco da Silva, et al. (2009).

2.3 Insertion sort

O Insertion Sort é de fácil implementação, similar ao Bubble Sort (SZWARCFITER e MARKENZON, 2015). É classificado como estável e simples, em seu funcionamento o algoritmo mantém uma sublista ordenada e sua execução se dá por comparação dos elementos e inserção direta nesta sublista já ordenada. À medida que o algoritmo percorre a lista de elementos, os elementos se organizam, um a um, em sua posição mais correta, onde o elemento a ser alocado terá, a sua esquerda um valor menor e à sua direita um valor maior. Em suma, o método consiste em comparar o i-ésimo item da sequência original a partir do i=2, com os Itens anteriores, estes itens anteriores compõem a sublista supracitada, em seguida adiciona o item na posição mais correta por meio de comparação e inserção, a inserção do elemento é efetuada movendo os itens para a direita durante a comparação e então inserindo-o na posição que ficou vazia. No processo de comparação e movimentação de itens existem duas condições de parada, quando o lugar mais correto é encontrado ou quando chegamos ao fim da sublista.

Pseudo-código do Insertion sort.

```
INSERTION(vetor):
    n = comprimento(vetor)
    para i=1, até n, i++:
        chaveatual = vetor[i]
        posicao = i

        enquanto posicao>0 e vetor[posicao-1]>chaveatual:
            vetor[posicao] = vetor[posicao-1]
            posicao = posicao-1
        vetor[posicao] = chaveatual
```

O Insertion sort é sensível a um conjunto de entradas inversamente ordenadas, tal situação resulta na comparação de cada elemento da lista original com cada elemento da sublista e inserção somente no fim da sublista. Seu melhor caso ocorre se o arranjo já está ordenado, fazendo somente uma comparação e uma inserção para cada interação. A complexidade deste algoritmo é O(n2). No anexo 1, encontra-se a análise de custo mais detalhada.

execução do insertion sort em ordenação crescente em um vetor de tamanho 8.

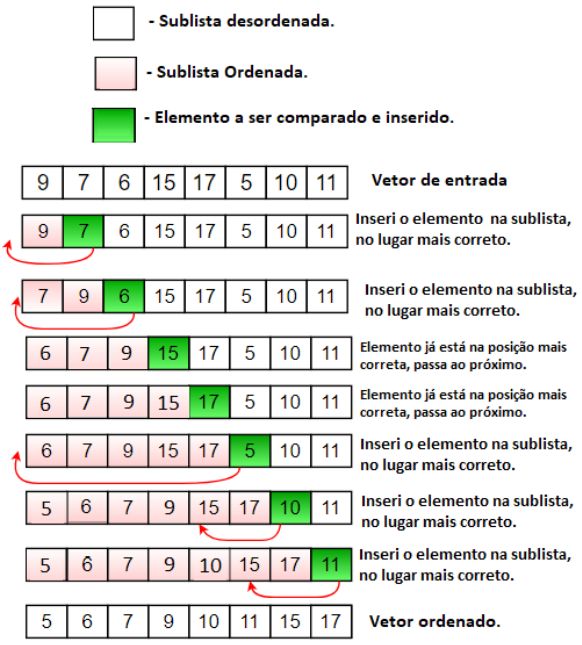

Figura 1. Representação de execução do Insertion sort.

Fonte: Geekforgeek (2020).

2.4 Quick sort

O quick sort é baseado na ideia de partir o arranjo em dois sub-arranjos, de tal forma que todos os elementos do primeiro arranjo sejam menores ou iguais a todos os elementos do segundo arranjo (HOARE, 1962). O algoritmo é recursivo e usa a famosa estratégia do dividir e conquistar, é classificado como eficiente e instável. A ideia é partir o problema de ordenar um conjunto com n itens em dois problemas menores. Essa partição tem início com a escolha de um elemento arbitrário do arranjo, este elemento é chamado de pivô, por convenção o pivô é atribuído como o último elemento da sequência de entrada. Depois, a sequência original é ordenada de modo que todos os elementos a direita do pivô são menores ou iguais a chave do pivô e todos os elementos a esquerda são maiores. Em seguida o algoritmo é aplicado recursivamente nos sub-arranjos que foram separadas pelo pivô. Os problemas menores são ordenados independentemente e depois os resultados são combinados para produzir a solução do problema maior.

Pseudo-código do Quick sort.

```
QUICKSORT(vetor, inicio, fim):
    se inicio<fim:
        pivô = PARTIR(vetor, inicio, fim)
        QUICKSORT(vetor, inicio, pivô-1)
        QUICKSORT(vetor, pivô+1, inicio)

PARTIR(vetor, inicio, fim):
    pivô = vetor[fim]
    i = inicio-1

    para j=inicio, até fim-1, j++:
        se vetor[j] <= pivô:
            i = i+1
            trocar vetor[i] <-> vetor[j]
    trocar vetor[i+1] <-> vetor[fim]
    retorne i+1
```

Uma característica interessante do quick sort é a sua ineficiência para arquivos já ordenados quando a escolha do pivô é inadequada. Por exemplo, a escolha sistemática dos extremos de um arquivo já ordenado leva ao seu pior caso (Nivio Ziviani, 2010).

O quick sort tem seu pior desempenho quando ocorre um particionamento ruim, o particionamento é considerado ruim quando o elemento pivô divide a sequência de forma desbalanceada, ou seja, divide em duas sublistas: uma com tamanho 0 e outra com

tamanho (n-1) no qual n se refere ao tamanho da sequência original. Este mau particionamento também ocorre quando o elemento pivô é o maior ou menor elemento da sequência. O melhor caso de particionamento acontece quando ele produz duas sublistas de tamanho não maior que n/2, uma vez que uma sublista terá tamanho [n/2] e outra tamanho [n/2] - 1. Nesse caso, o quick sort é executado com maior rapidez. A complexidade deste algoritmo é O(n2). No anexo 1, encontra-se a análise de custo mais detalhada.

execução do Quick sort em ordenação crescente em um vetor de tamanho 9.

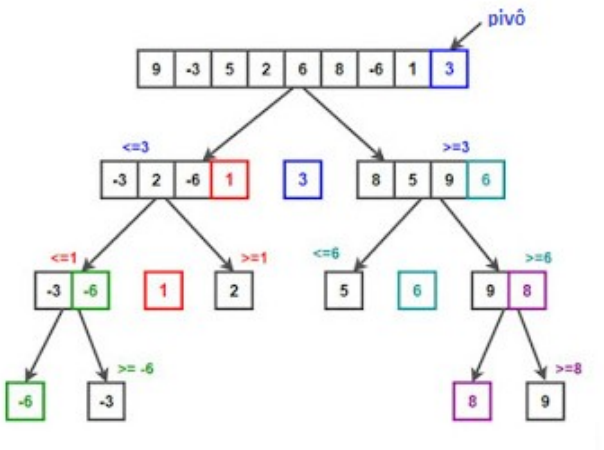

Figura 3. Representação de execução do Quick sort.

Fonte: Androp Notes (2020).

2.5 Literatura

Trabalhos similares existem na literatura – (DAVID COUTO BITENCOURT, 2014), (Cristino Divino de Freitas Júnior, et al. 2014), (Jackson É. G. Souza, et al. 2017), este trabalho é alternativo aos existentes e tem os mesmos como referência.

3. Procedimento

3.1 Material

O trabalho tem como material de estudo os algoritmos bubble sort, insertion sort e quick sort. A implementação dos algoritmos foi feita na linguagem de programação python e o modelo de implementação de cada algoritmo foi escolhido conforme o modelo mais clássico

e mais usado. A escolha desta linguagem de programação se deu por ser uma linguagem de fácil leitura e por o código exigir poucas linhas se comparado ao mesmo programa em outras linguagens. (a implementação dos algoritmos encontram-se nos anexos 2, 3 e 4).

3.2 Ferramentas

As ferramentas utilizadas no trabalho foram:

- 3 bases de dados que foram geradas por um script em python (apêndice 1). Cada base de dados contém 83 vetores sendo 50 com 500 elementos, 30 com 1000 elementos e 3 com 10000 elementos.
- Um script que executa os testes (apêndice 2).
- Bibliotecas e pacotes padrões do python.
- IDE Visual Studio Code.
- Um computador com as seguintes características: processador Intel Celeron 1.80GHz, 4 GB de memória RAM e disco rígido de 500 GB, com sistema operacional Ubuntu 20.04 LTS.

3.3 Método

No desenvolvimento deste trabalho foi seguido as seguintes etapas:

Primeiro foi feito uma pesquisa e um levantamento de artigos e livros relacionados ao tema. Em seguida foi feita a escolha dos algoritmos e foi definido o modelo de implementação de cada algoritmo, o critério de escolha dos algoritmos foi a popularidade e complexidade de implementação, sendo assim escolhidos os algoritmos mais populares e de implementação mais simples.

Em seguida foi executada uma análise para verificar a eficácia dos algoritmos e uma pesquisa na literatura existente para descobrir em quais situações os algoritmos são mais sensíveis. E foi descoberto quais circunstâncias levam os algoritmos ao seu pior, melhor e médio caso de complexidade de crescimento.

Por vez, foram definidos os critérios de avaliação dos algoritmos, esses foram escolhidos baseados nos parâmetros mais importantes que se deve considerar na escolha de um algoritmo de ordenação, que são: a complexidade de crescimento, a complexidade de comparações e trocas, além do tempo de execução aplicando testes reais.

Adiante foi definida as bases de dados usadas, a definição foi baseada na análise dos algoritmos feita anteriormente, as bases tem características sobre os pontos em que os algoritmos são mais sensíveis. Todas as 3 bases de dados contém 83 vetores com tamanhos de 100, 1000 e 10000 elementos, Uma delas tem vetores somente ordenados em ordem crescente, outra contém somente vetores ordenados em ordem decrescente e a última contém somente vetores com elementos em ordem aleatória. Foi definido que os elementos a serem ordenados seriam somente números inteiros para facilitar a execução dos testes e compreensão dos resultados.

Em seguida foi definida a linguagem de programação para implementar os algoritmos e os demais scripts necessários. Pós isso foi desenvolvido um código para automatizar o processo de testes. E por vez, foi definido onde realizar os testes.

Por fim os scripts foram implementados e testados para garantir a eficácia. Pós implementação e com todas as ferramentas necessárias prontas, foi executado o processo de testes e catalogado os dados obtidos e esses dados foram sintetizados e transformados nos resultados.

3.4 Execução

A realização dos testes foi feita da seguinte forma:

Para cada base de dados foi executado o script (apêndice 2) que gera a base e aplica cada um dos 3 algoritmos na mesma, os testes iniciaram com vetores com capacidade para 500 elementos, são avaliados os tempos de execução, os números de comparações e, os números de trocas em cada base de dados. O mesmo procedimento é realizado com vetores de tamanhos 1.000 e 10.000, com os resultados espera-se apontar as vantagens e desvantagens ao se fazer uso de um desses algoritmos. O script que executa os testes armazena os resultados obtidos da aplicação em arquivos de texto. Esses arquivos com os resultados foram estudados, analisados e sintetizados nas tabelas presentes na seção 4.

Todos os algoritmos de ordenação foram aplicados sobre as mesmas bases de dados, validando assim o desempenho de cada um. Cada um deles foi estudado em detalhes para verificar sua forma de execução.

Os critérios de avaliação escolhidos para comparar os algoritmos foram a complexidade do algoritmo, a quantidade de comparações e trocas e o tempo gasto na execução dos testes.

4. Resultados

4.1 Resultados em vetores com 500 elementos.

Na tabela 1, encontram-se os resultados obtidos da aplicação dos 3 algoritmos nas bases de dados ordenada, inversamente ordenada e aleatória.

Tabela 1 - Valores médios do desempenho dos algoritmos em vetores de tamanho 500.

Vetores com 500 elementos									
Ordem	Crescente			Inversa			Aleatória		
Algoritmo	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas
Bubble sort	0.0153330329	124750.0	0.0	0.0359970887	124750.0	124750.0	0.0265488097	124750.0	62408.7
Insertion sort	0.0002108198	499.0	0.0	0.0276873538	499.0	124750.0	0.0145317853	499.0	62408.7
Quick sort	0.0292628071	124750.0	125249.0	0.0196120072	124750.0	62749.0	0.0012106330	4847.8	2708.1

Na tabela pode-se observar que em uma lista já ordenada, o insertion sort foi o algoritmo com melhor desempenho. Seguido do bubble sort, que apesar de ter o mesmo número de comparações que o quick sort não teve nenhuma troca. Esta quantidade de trocas elevada do quick sort se dá devido a todos os elementos do vetor serem menores que o pivô, que foi definido como o ultimo elemento da lista, esta escolha leva o quick sort ao seu pior caso de complexidade de crescimento.

Em um vetor inversamente ordenado é evidente a ineficiência do bubble sort, que vai ao seu pior caso de crescimento com um alto número de comparações e trocas. Em relação ao número de comparações podemos destacar a eficiência do insertion sort e a ineficiência do quick sort.

Em um vetor de ordem aleatória podemos destacar a eficiência do quick sort, que obteve o melhor desempenho com o menor número de trocas. Seguido do insertion sort com o segundo melhor desempenho e do bubble sort.

4.2 Resultados em vetores com 1000 elementos.

Os resultados obtidos da aplicação dos 3 algoritmos nas bases de dados com vetores de tamanho 1000, encontram-se na tabela 2.

Tabela 2 - Valores médios do desempenho dos algoritmos em vetores de tamanho 1000.

Vetores com 1000 elementos									
Ordem	Crescente			Inversa			Aleatória		
Algoritmo	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas
Bubble sort	0.0641962407	499500.0	0.0	0.1498441605	499500.0	499500.0	0.1111475495	499500.0	251320.1
Insertion sort	0.0004509884	999.0	0.0	0.1175085791	999.0	499500.0	0.0613152167	999.0	251320.1
Quick sort	0.1185840656	499500.0	500499.0	0.08146436893	499500.0	250499.0	0.0026849382	11082.2	6273.1

Assim como na tabela 1, em vetores com ordenação crescente pode-se ver a ineficiência do quick sort e o bom desempenho do insertion sort. Do mesmo modo, em vetores inversamente ordenados, podemos ver que o insertion sort juntamente com o quick sort continuam apresentando desempenho superior ao bubble sort. Quanto aos vetores

aleatórios, o quick sort mantém a melhor performance, assim como, o Bubble Sort permanece com altos valores de comparações e trocas.

4.3 Resultados em vetores com 10000 elementos.

Na tabela 3 encontra-se os resultados obtidos da aplicação em vetores de tamanho 10000.

Tabela 3 - Valores médios do desempenho dos algoritmos em vetores de tamanho 10000.

Vetores com 10000 elementos									
Ordem	Crescente			Inversa			Aleatória		
Algoritmo	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas	Tempo (s)	Nº Comp.	Trocas
Bubble sort	7.0762577376	49995000.0	0.0	15.9315021770	49995000.0	49995000.0	11.766903072	49995000.0	25001998.0
Insertion sort	0.0045343540	9999.0	0.0	12.6689659200	9999.0	49995000.0	6.3798981683	9999.0	25001998.0
Quick sort	12.0772303606	49995000.0	50004999.0	8.1006043210	49995000.0	25004999.0	0.0353241073	151978.3	88351.0

Mais uma vez o padrão se repete, o insertion sort obtém o melhor desempenho em vetores ordenados, o quick sort manifesta uma vantagem considerável em tempo de execução sobre o insertion sort em vetores inversamente ordenados, e em vetores aleatórios o quick sort obtém uma performance superior aos demais.

5. Conclusão

O algoritmo bubble sort apresenta resultados insatisfatórios na maioria dos testes, com ênfase no alto valor de comparações, isto pode ser traduzido como um grande consumo de processamento, o que pode resultar em lentidão. É o algoritmo de mais fácil implementação, porém na opinião da autora é recomendado somente para usos educacionais ou para pequenos projetos em que os vetores terão tamanhos pequenos, visando que sua aplicação em projetos grandes tornariam o projeto ineficiente.

O algoritmo insertion sort, obteve resultados satisfatórios somente em vetores pequenos, portanto na opinião da autora ele é útil para ordenação de estruturas lineares pequenas, principalmente quando sabe-se que a estrutura pode estar previamente ordenada, o que leva o algoritmo ao seu melhor caso de complexidade de crescimento. Sua principal vantagem é o pequeno número de comparações, e, o excessivo número de trocas, sua desvantagem.

Por sua vez, o algoritmo quick sort obteve resultados satisfatórios em vetores com tamanhos grandes. O algoritmo ao subdividir o vetor e fazer inserções diretas utilizando o pivô, reduz seu tempo de execução, mas, as quantidades de comparações e principalmente, trocas ainda são muito altas. Apesar disso, na opinião da autora o quick sort se apresenta uma boa opção para projetos de grande porte e quando sabe-se que a estrutura a ser ordenada estará em ordem aleatória o que leva o algoritmo a sua melhor performance, e para situações em que o objetivo é a execução em um menor tempo, mesmo que para isso haja um maior consumo de recursos computacionais de processamento.

Por fim, pode-se concluir que em relação a ordenação de estruturas pequenas ou estruturas previamente ordenadas o insertion sort é o algoritmo mais eficiente. E quanto a estruturas grandes ou aleatórias o quick sort é a melhor opção. O bubble sort não é recomendado para uso em nenhuma dessas situações.

Referências

BRADLEY N. MILLER; DAVID L. RANUM. Título: Problem Solving with Algorithms and Data Structures using Python,2013. Disponível em: https://www.cs.auckland.ac.nz/compsci105s1c/resources/ProblemSolvingwithAlgorithmsandDataStructures.pdf. Acesso: 10/09/2020

Prof. JAIRO FRANCISCO DE SOUZA, Apostila de Estrutura de Dados II. Disponível em: https://www.ufjf.br/jairo_souza/files/2009/12/2-Ordena%c3%a7%c3%a3o-BubbleSort.pdf. Acesso: 10/09/2020.

Prof. GILBERTO AMADO DE A. C. FILHO, Apostila de Introdução a programação. Disponível em: https://drive.google.com/file/d/0B3ouk05r7UFPcWdlN3ZXM095VG8/view. Acesso: 11/19/2020.

CORMEN, THOMAS H. et al. Título: Algoritmos: Teoria e prática. 3ª Edição. Elsevier, 2012. Disponível em: https://computerscience360.files.wordpress.com/2018/02/algoritmos-teoria-e-prc3a1tica-3ed-thomas-cormen.pdf. Acesso: 12/09/2020.

PARREIRA JÚNIOR, Walteno M. Apostila de Análise de Algoritmos. Ituiutaba - MG: FEIT-UEMG, 2012. Diponível em: http://waltenomartins.com.br/ap_aa_v1.pdf . Acesso: 12/09/2020.

Júnior, Cristino Divino de Freitas. Et al. Artigo sobre Metodos de ordenação: a importância da escolha do método correto. Intercursos, v. 13, n.1, Jan-Jun. 2014 – ISSN 2179-9059. Disponível em: http://www.waltenomartins.com.br/intercursos_v13n1a.pdf. Acesso: 12/09/2020.

Szwarcfiter, J. L. and Markezon, L. (2015). “Estruturas de Dados e Seus Algoritmos.” 3ª edição. Rio de Janeiro. LTC.

HOARE, C. A. R. Quicksort. The computer journal, British Computer Society, v. 5, n. 1, p. 10-16, 1962.

Geeg for geek, Insertion sort. Disponível em: https://www.geeksforgeeks.org/recursive-insertion-sort/. Acesso: 13/09/2020.

Treina web, conheça principais algoritmos de ordenação. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/. acesso: 13/09/2020.

Ziviani, Nivio. Projeto de Algoritmos com Implementações em Pascal e C - 3ª Ed. 2010

Disponível em: https://docero.com.br/doc/n80sce

Acesso: 13/09/2020.

Silva, Glauco. et al. Artigo sobre Análise comparativa de métodos de ordenação, 2009. Disponível em: https://www.researchgate.net/publication/308900585_Analise_comparativa_de_metodos_de_ordenacao. Acesso: 13/09/2020.

Magalhães, Lúcia Helena. Et al. artigo sobre análise da complexidade de algoritmos de ordenação. 2014 Disponível em: https://www.academia.edu/10859131/ANÁLISE_DA_COMPLEXIDADE_DE_ALGORITMOS_DE_ORDENAÇÃO. Acesso: 14/09/2020.

Bitencourt , David Couto. Artigo sobre Análise dos métodos de ordenação, 2014. Disponível em: https://www.academia.edu/7714659/Análise_dos_métodos_de_ordenação. Acesso: 15/09/2020.

Souza, Jackson É. G. et al. Artigo sobre Algoritmos de Ordenação: Um Estudo Comparativo, 2017. Diponível em: https://periodicos.ufersa.edu.br/index.php/ecop/article/download/7082/6540. Acesso: 15/09/2020.

Anexos

Anexo 1 - Analise dos algoritmos.

Algoritmo	**Comparações**			**Trocas**			**Complexidade**
	Melhor caso	**Caso médio**	**Pior caso**	**Melhor caso**	**Caso médio**	**Pior caso**	
Bubble sort	$O(n^2)$			$O(n^2)$			$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(n^2)$

Quick sort	O(n logn)	O(n²)	-	O(n²)

Anexo 2 – Código fonte do Bubble sort.

```
def bubblesort(alist):

n = len(alist)
for j in range(n-1,0,-1):
        for i in range(j):
                if alist[i] > alist[i+1]:
                        alist[i], alist[i+1] = alist[i+1], alist[i]
```

Anexo 3 – Código fonte do Insertion sort.

```
def insertionsort(alist):

        for ind in range(1,len(alist)):
                currentvalue = alist[ind]
                position = ind

                while position > 0 and alist[position-1]>currentvalue:
                        alist[position] = alist[position-1]
                        position = position-1

                alist[position] = currentvalue
```

Anexo 4 – Código fonte do Quick sort.

```
def quickSort(arr):

        quick_Sort(arr,0,len(arr)-1)
def quick_Sort(arr,low,high):
        if low < high:
                pi = partition(arr,low,high)
                quick_Sort(arr, low, pi-1)
                quick_Sort(arr, pi+1, high)

def partition(arr,low,high):
        i = ( low-1 )
        pivot = arr[high] #pivô
        for j in range(low , high):
                if arr[j] <= pivot:
                i = i+1
                arr[i],arr[j] = arr[j],arr[i]
        arr[i+1],arr[high] = arr[high],arr[i+1]
        return ( i+1 )
```

Apêndices

Apêndice 1 – Código fonte que gera as bases de dados.

```
import shelve, random


def CreateDBAleatoria():
	dataBase = shelve.open('DBAleatoria')
	for num in range(1,50+1):
		vetor = [random.randint(0,10000) for _ in range(500)]
		dataBase[str(num)] = vetor

	for num in range(51,80+1):
		vetor = [random.randint(0,10000) for _ in range(1000)]
		dataBase[str(num)] = vetor

	for num in range(81,83+1):
		vetor = [random.randint(0,10000) for _ in range(10000)]
		dataBase[str(num)] = vetor
	dataBase.close()

def CreateDBOrdenada():
	dataBase = shelve.open('DBOrdenada')

	i,j =1,501
	for num in range(1,50+1):
		vetor = [ele for ele in range(i,j)]
		dataBase[str(num)] = vetor
		i,j = j, j+500

	i,j = 1,1001
	for num in range(51,80+1):
		vetor = [ele for ele in range(i,j)]
		dataBase[str(num)] = vetor
		i,j = j,j+1000

	i,j = 1,10001
	for num in range(81,83+1):
		vetor = [ele for ele in range(i,j)]
		dataBase[str(num)] = vetor
		i,j = j,j+10000
	dataBase.close()

def CreateDBInversa():
	dataBase = shelve.open('DBInversa')

	i,j = 0,500
	for num in range(1,50+1):
		vetor = [ele for ele in range(j,i,-1)]
		dataBase[str(num)] = vetor
```

```
            i,j = j,j+500
        i,j = 0,1000
        for num in range(51,80+1):
            vetor = [ele for ele in range(j,i,-1)]
            dataBase[str(num)] = vetor
            i,j = j,j+1000

        i,j = 0,10000
        for num in range(81,83+1):
            vetor = [ele for ele in range(j,i,-1)]
            dataBase[str(num)] = vetor
            i,j = j,j+10000
        dataBase.close()
```

Apêndice 2 – Código que executa os testes.

```
from time import process_time

import BubbleSort, InsetionSort, QuickSort, CriandoDB, shelve

def Escreve(alg,tbase, dado, aux='CompTroca'):
    arquivo_out = open(f'{alg}_{tbase}.txt','a')

    if aux != 'tempo':
        arquivo_out.write(f'Número médio de comparações do {alg} com base {tbase} em vetores de:\
        n500 elementos:\n')
        arquivo_out.write(f'comparações: {dado[0][0]:.1f} Trocas: {dado[0][1]:.1f}\n')
        arquivo_out.write(f'1000 elementos:\n')
        arquivo_out.write(f'comparações: {dado[1][0]:.1f} Trocas: {dado[1][1]:.1f}\n')
        arquivo_out.write(f'10000 elementos:\n')
        arquivo_out.write(f'comparações: {dado[2][0]:.1f} Trocas: {dado[2][1]:.1f}\n')
    else:
        arquivo_out.write(f'\nTempo médio da ordenação do {alg} com base {tbase} em vetores de:\
        n500 elementos:\n')
        arquivo_out.write(f'{dado[0]}\n')
        arquivo_out.write(f'1000 elementos:\n')
        arquivo_out.write(f'{dado[1]}\n')
        arquivo_out.write(f'10000 elementos:\n')
        arquivo_out.write(f'{dado[2]}\n')
    arquivo_out.close()


def Conta_CT(DB, alg):
    resultado = []

    comp = 0
    trocas = 0
    for i in range(1,50+1):
        i = str(i)
        c, t = alg(DB[i])
```

```
                comp += c
                trocas += t
        resultado.append((comp/50, trocas/50))

        comp = 0
        trocas = 0
        for i in range(51,80+1):
                i = str(i)
                c, t = alg(DB[i])
                comp += c
                trocas += t
        resultado.append((comp/30, trocas/30))

        comp = 0
        trocas = 0
        for i in range(81,83+1):
                i = str(i)
                c, t = alg(DB[i])
                comp += c
                trocas += t
        resultado.append((comp/3, trocas/3))

        return resultado


def main():
        base = input('Base: ').strip().capitalize()

        if base == 'Ordenada':
                CriandoDB.CreateDBOrdenada() #Cria o data base
                DataBase = shelve.open('DBOrdenada')

        elif base == 'Inversa':
                CriandoDB.CreateDBInversa() #Cria o data base
                DataBase = shelve.open('DBInversa')
        elif base == 'Aleatoria':
                CriandoDB.CreateDBAleatoria() #Cria o data base
                DataBase = shelve.open('DBAleatoria')

        valores_comp_trocas_b = Conta_CT(DataBase, BubbleSort.bubblesort_regitros)
        Escreve('bubble',base,valores_comp_trocas_b)
        tempo = BubbleSort.Calcula_Tempo_Bubble(DataBase)
        Escreve('bubble',base,tempo,"tempo")
        valores_comp_trocas_i = Conta_CT(DataBase, InsetionSort.insertionsort_registros)
        Escreve('insertion',base,valores_comp_trocas_i)
        tempo = InsetionSort.Calcula_Tempo_insertion(DataBase)
        Escreve('insertion',base,tempo,"tempo")

        valores_comp_trocas_q = Conta_CT(DataBase, QuickSort.quickSort_registros)
        Escreve('quick',base,valores_comp_trocas_q)
```

```
        tempo = QuickSort.Calcula_Tempo_Quick(DataBase)
        Escreve('quick',base,tempo,"tempo")

        DataBase.close()
        return None

if __name__ == '__main__':
        main()
```