UTFPR
Engenharia de Computação
Tópicos em Projeto e Análise de Algoritmos

Relatório Análisando Algoritmos de Ordenação

Alunos: Guido Margonar Moreira

Junho
2023

Conteúdo

1 Introdução

Ao longo da disciplina de Tópico em Análise e Projetos de Algoritmos foram estudados diferentes métodos para análise da complexidade de algoritmos, sendo a ênfase principal no cálculo da complexidade de tempo visto que o custo de memória costuma ser um problema menor que pode ser resolvido ou aumentando o custo de tempo ou acoplando mais unidades de memória ao Hardware.

A fim de colocar a testes os conhecimentos estudados e aprender mais sobre certos métodos foram implementados 4 algoritmos ordenadores de vetores, sendo eles Insertion Sort, Merge Sort, QuickSort e Radix Sort, para que a pós a implementação de cada um fosse realizada uma série de testes computando seus desempenho na complexidade de tempo.

2 Metodologia

Para a implementação e análise dos algoritmos foi escolhido o python, linguagem de programação de alto nível conhecida por sua simplicidade, versatilidade, apesar de ter desempenho lento em processos mais pesados, o que de certa forma é uma vantagem para a análise da complexidade de tempo visto que as diferenças de desempenho nos casos muito rápidos ficara um pouco mais clara já que os processos levaram mais tempo mas seguindo a mesma proporção. O sistema operacional utilizado para os algoritmos que precisavam de mais memória ou duravam muito tempo foi o Linux, porém alguns algoritmos foram executados online por meio da IDE do site Repl.it. já que os códigos eram desenvolvidos diretamente nela e ela facilita salvar os dados online automaticamente.

As bibliotecas do python utlizadas para implementar e analisar os algoritmos foram **time**, para pegar a contagem do tempo entre o inicio e o fim das execuções dos algoritmos permitindo calcular o tempo gasto, **random**, para gerar valores aleatórios que preenchem os vetores no caso em que o vetor é aleatório (em alguns gráficos chamado de caso médio) e **matplotlib.pyplot**, uma ferramenta da matplotlib que permite a criação de gráficos ao passar os vetores e informações da formatação para a função **.plot** que podem depois serem salvos como um arquivo .png.

O primeiro Algoritmo implementado(originalmente em C e depois adaptado ao python para ficar no mesmo padrão dos demais) foi o Insertion Sort ele funciona começando no segundo elemento e comparando com os elementos já ordenados antes dele, passando os maiores para a direita sendo inserido ao encontrar um valor menor que ele próprio, assim ele passa por todo o vetor organizando um elemento por iteração, no pior caso onde todos os vetores estiverem ordenados ao contrário do desejado ele tem complexidade $O(n^2)$ visto que precisaria mover todos os movimentos já ordenados para a direita para cada um dos n elementos.

Já o algoritmo Merge Sort funciona por meio da divisão do vetor principal em dois recursivamente até que se chega em vetores unitários que são então ordenados pela função **Combina** que ordena os vetores comparando as duas metades passadas assim as divisões são unidas recurisvamente até o vetor interio estar ordenado, assim a complexidade do algoritmo é $T(n) = 2 * T(n/2) + O(n)$ que por meio do teorema mestre equivale a $O(n \log n)$.

Em seguida foi implementado o algoritmo Quicksort, ele funciona por meio da escolha de um pivô(neste caso pegamos o valor que está no meio do vetor passado para a função) que é usado para dividir o vetor em 2 sendo que os valores menores ficam no vetor da esquerda e os maiores no da direita, os novos subvetores gerados são enviados para uma chamada recursiva da

função que repete o processo com um novo pivô até que todos os subvetores unidos tenham ordenado o vetor inteiro, o pior caso para esse algoritmo seria quando o pivô escolhesse sempre o elemento de menor ou maior valor dependendo do estado do vetor o que faria com que ele operasse igual a um Insertion sort(Exemplo: pega sempre o maior valor valor mas o vetor está ordenado de trás pra frente então sempre vai passar todos os elementos para a esquerda gerando um subvetor de tamanho n-1 e outro de tamanho 0) tendo assim complexidade $O(n^2)$ para o pior caso mas isso é bem improvável e normalmente para divisões na média que geram 2 subvetores a complexidade é $O(n \log n)$.

Por fim o Radix Sort que funciona analisando os dígitos dos elementos que serão organizados começando pelos dígitos das unidades, indo para dezenas, depois centena até a casa dos dígitos do maior elemento do vetor, os elementos do vetor são ordenados em cada iteração baseado nos seus índices indo de 0 a 9, para organizar-los o algoritmos cria um vetor do mesmo tamanho do original e insere os elementos nele baseado no valor do digito, esse processo tem custo $O(n)$ e será executado um número de vezes igual ao número de dígitos do maior elemento do vetor, dessa maneira a proporção fica igual a $m * Ndigitosmaiorelemento$ mas como esse número de dígitos sobe numa taxa muito baixa acaba que a função fica equivalente a multiplicar uma constante por n (exemplo para n = 1000000000 teríamos $O(10 * n)$) que equivale simplesmente a complexidade $O(n)$, vale lembrar que para alcançar isso o algoritmo usa bem mais da complexidade de espaço alocando memória para um vetor auxiliar do mesmo tamanho do original.

3 Resultados

A implementação de todos os algoritmos foi bem sucedida porém alguns algoritmos tem em sua legenda o "caso médio"para o caso do vetor ordenado com elementos aleatórios que nem sempre é o caso médio (como se pode observar nos gráficos 2 e 3). Como é possível observar no resultados o Insertion Sort obteve sua complexidade $O(n^2)$ nos pior e médio casos, como se pode observar no 1. O gráfico 2 mostra que o mergeSort apresentou um comportamento quase que linear como é esperado de sua complexidade $O(n \log n)$. No caso do QuickSort podemos observar em 3 que o caso com vetor ordenados ao contrário eventualmente ficou pior do que o de vetores aleatórios mas todos os casos seguiram a complexidade $O(n \log n)$ esperada. Por fim, o radix apresentou comportamento linear embora o caso de vetores aleatórios tenha saído um pouco da curva (possivelmente algum problema de desempenho relacionado com o uso da biblioteca random).

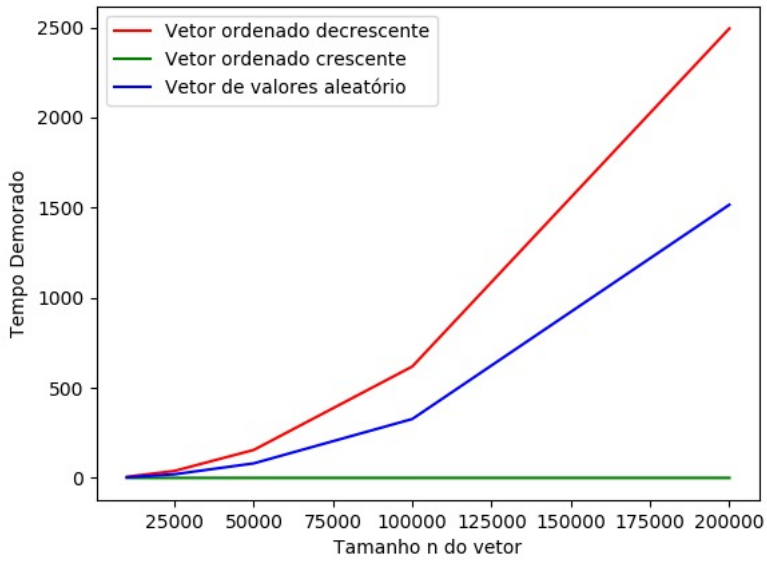

Figura 1: Gráfico desempenho Merge Sort. Fonte: Autoria própria.

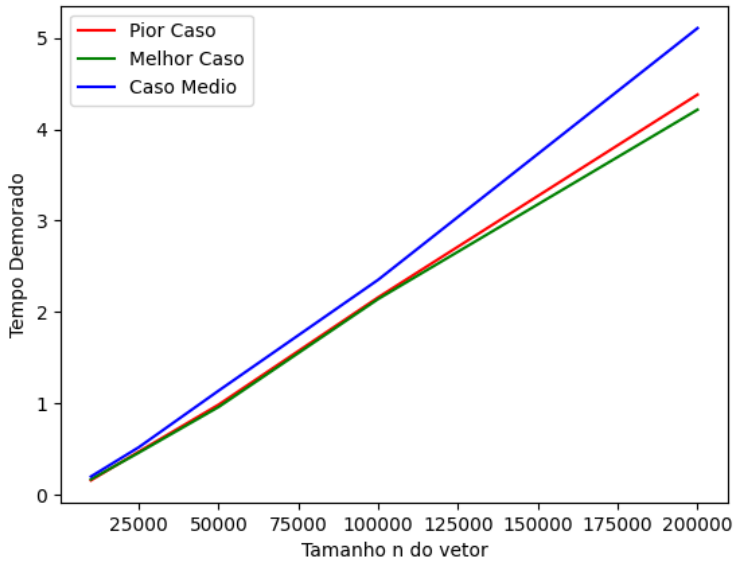

Figura 2: Gráfico desempenho Insertion Sort. Fonte: Autoria própria.

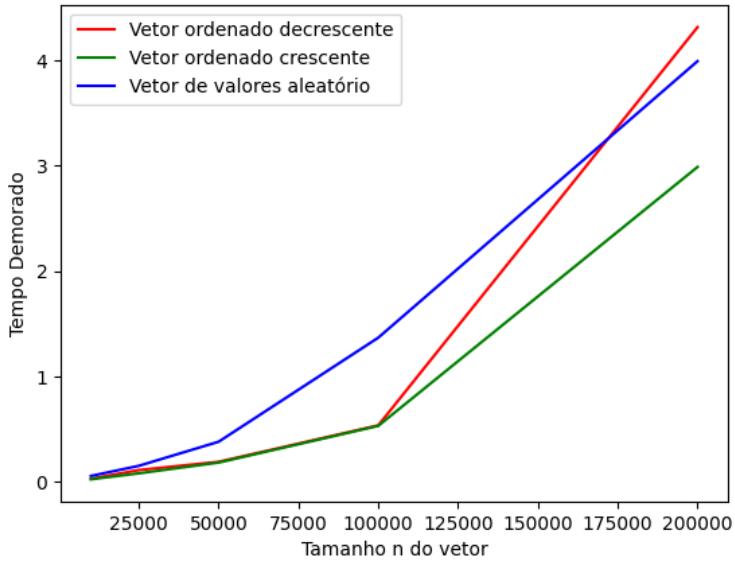

Figura 3: Gráfico desempenho QuickSort. Fonte: Autoria própria.

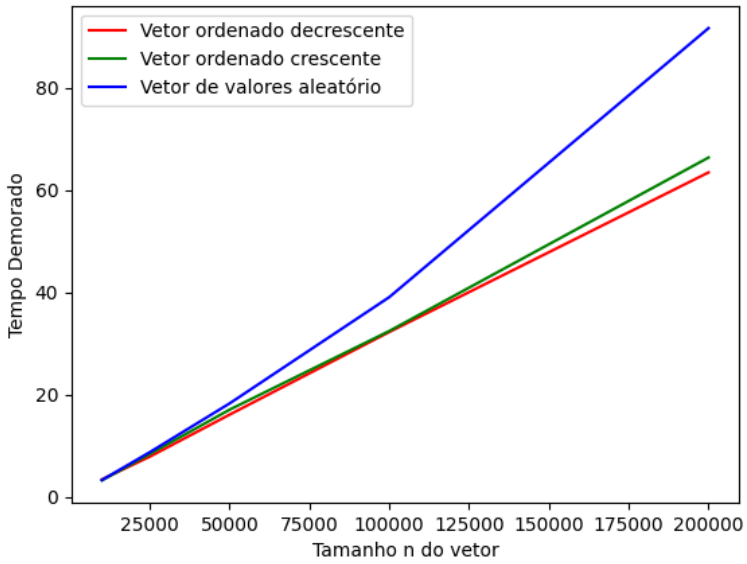

Figura 4: Gráfico desempenho Radix Sort. Fonte: Autoria própria.

Para propósitos de comparação nas tabelas 1 e 2 estão reunidos os tempos tomados para execução de cada algoritmo nos testes com vetores já ordenados e ordenados de trás pra frente.

Na análise dos vetores ordenados ao contrário da tabela 1 é possível observar que apesar do comportamento linear para os vetores com os tamanhos analisados o radix sort, mesmo se mantendo linear (o que fica claro conforme o tempo aumenta na mesma proporção do tamanho n do vetor), ainda ficou para trás do quick e merge (embora para vetores muito maiores o esperado é que o radix os ultrapasse devido a sua complexidade linear), já o InsertionSort como esperado apresentou o pior resultado com sua complexidade quadrática chegando a demorar uma média de mais de 41 minutos para o vetor de tamanho n = 200000.

	n = 10000	n = 25000	n = 50000	n = 100000	n = 200000
Insertion Sort	6.192871332168579	38.83845808506012	155.33930759429933	618.4095521688462	2494.200502443314
Merge Sort	0.1576008081436157	0.47170252799987794	0.9864107131958008	2.1612355709075928	4.379689240455628
QuickSort	0.030090808868408203	0.11224687099456787	0.19073803424835206	0.5365639686584472	4.316582036018372
Radix Sort	3.389008378982544	7.843920612335205	16.06347460746765	32.22314372062683	63.43470788002014

Tabela 1: Tempos médios (em segundos) para vetor Ordenado de trás pra frente.

Já na tabela 2 quando os algoritmos recebem um vetor já ordenados podemos notar que o Merge e o Radix permaneceram praticamente inalterados

visto que realizaram aproximadamente o mesmo número de operações, enquanto o QuickSort teve um pequeno ganho de desempenho mais notável no n = 20000, e o InsertionSort conseguiu ser mais rápido de todos visto que ele só faz uma comparação para cada elemento no vetor percebendo que não precisa inseri-lo em outro lugar assim conseguindo o melhor desempenho.

	n = 10000	n = 25000	n = 50000	n = 100000	n = 200000
Insertion Sort	0.001297140121459961	0.003203010559082031	0.006330752372741699	0.01273491382598877	0.025783801078796388
Merge Sort	0.16856138706207274	0.46036405563354493	0.9604602336883545	2.1427748918533327	4.214479875564575
QuickSort	0.024396991729736327	0.08121931552886963	0.18353519439697266	0.531926941871643	2.989546060562134
Radix Sort	3.2110217094421385	8.360768103599549	17.039757132530212	32.4150808095932	66.34729266166687

Tabela 2: Tempos médios (em segundos) para vetor já ordenado.

4 Conclusão

Por meio dos dados analisados foi possível concluir que para os tamanhos de vetores analisados o s algoritmos MergeSort e QuickSort seriam os mais recomendados, sendo o MergeSort o melhor para se manter consistente (ao menos com o método que foi utilizado para pegar o pivô), os dados também mostraram que para vetores maiores eventualmente o Radix Sort será mais rápido embora venha com a desvantagem de que para esses casos ele vai precisar dobrar o uso de memória visto que vai criar um vetor auxiliar do mesmo tamanho do original, e os testes mostraram que o InsertionSort apesar de dar conta das situações de vetores pequenos ele já se mostrou menos eficiente desde o primeiro caso ficando progressivamente pior chegando a quase quase 42 minutos no pior caso porém se mostrou o mais eficiente caso não houvesse necessidade de organizar o vetor ao contrário dos demais, além de ser o com menor uso de memória visto que não a chamadas recursivas e tudo é feito diretamente no vetor original. Assim os algoritmos possuem seus nichos de aplicações não existindo um absolutamente superior dentre eles, embora na média alguns tenham melhor tido desempenho do que os outros a melhor escolha ainda depende do ambiente em que se está trabalhando, tamanho do vetor, tempo aceitável para execução e da capacidade de memória disponível.

Referências

Radix Sort (With Code in Python, C++, Java and C) - Programiz. 2020. Disponível em: `https://www.programiz.com/dsa/radix-sort`. Acesso em: 06 de Junho de 2023.