UNIVERSIDADE FEDERAL DO PIAUÍ
CAMPUS SENADOR HELVÍDIO NUNES DE BARROS
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO
DISCIPLINA: PROJETO E ANÁLISE DE ALGORITMOS
DOCENTE: ISMAEL DE HOLANDA LEAL

ALGORITMOS DE ORDENAÇÃO

Alex William Leal

Mateus Pinto Garcia

Tiago De Moura Oliveira

PICOS – PI

15 de Outubro de 2019

1. Introdução

A ordenação ou classificação de registros consiste em organizá-los em ordem crescente ou decrescente, e assim facilitar a recuperação dos dados, ou seja, auxiliar nas buscas e pesquisas de ocorrências de determinado elemento em um conjunto ordenado, de modo mais eficiente. Por exemplo, o tempo de pesquisa de um número em um catálogo telefônico seria menor, se os nomes das pessoas estivessem em ordem alfabética.

As ordens do algoritmo de ordenação são facilmente definidas, seja por ordem numérica ou ordem alfabética, porém existem ordens que podem ser não tão comuns de se estabelecer, principalmente de dados compostos.

Entretanto, para melhor escolha de um método de ordenação é necessário saber de onde são os dados que serão processados. Classificando o tempo de acesso a um elemento, e a possibilidade de acesso direto a um elemento.

O tempo de acesso a um elemento é a complexidade necessária para acessar um elemento em uma estrutura. Já a possibilidade de acesso direto é a capacidade ou impossibilidade de acessar um elemento diretamente na estrutura.

Para classificarmos estes dois ambientes, utilizamos o meio em que está armazenado os dados. Designado por ordenação interna, que é quando queremos ordenar informações em memória, e ordenação externa, que é quando queremos ordenar informações em arquivo.

2. Algoritmos

Os três algoritmos de ordenação implementados e citados neste trabalho serão, o *QuickSort*, *BucketSort* e *HeapSort*.

2.1 *QuickSort*

Criado por Tony Hoare, o *QuickSort* foi desenvolvido a partir da tentativa de traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido.

O *QuickSort* é um método de ordenação baseado no conceito de dividir-e-conquistar. Primeiramente ele seleciona um elemento no qual é chamado de pivô, depois remover este pivô e

particiona os elementos restantes em duas sequências, uma com os valores menores do que o pivô e outras com valores maiores. Tudo isso de forma recursiva.

O tempo de execução deste algoritmo depende se o particionamento é ou não balanceado. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Porém, se o particionamento não é balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção.

Definimos sua complexidade em melhor caso, caso médio e pior caso. O pior caso ocorre quando o pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sublistas. Se isso acontece em todas as chamadas do método de particionamento, então cada etapa recursiva chamará listas de tamanho igual à lista anterior -1. Assim, teremos a seguinte relação de recorrência:

$$T(n) = T(n-1) + T(0) + \theta(n)$$

$$T(n-1) + \theta(n)$$

Imagem 1: Custo de pior caso *QuickSort*

Se a cada nível de recursão, somarmos os custos, teremos o valor da equação 1, portanto o algoritmo terá o tempo de execução igual à equação 1.

$$\Theta n^2$$

O melhor caso para o *QuickSort* é ocorrido quando ele produz duas listas de tamanho não maior que n/2, uma vez que a lista terá tamanho [n/2] e a outra tamanho [n/2] - 1. Assim o algoritmo é executado com maior rapidez. Sua recorrência é a seguinte:

$$T(n) \leq 2T(\frac{n}{2}) + \theta(n)$$

Figura 2: Custo do melhor caso *QuickSort*

A partir do teorema mestre, a solução T(n) será a seguinte equação 2.

$$O(nlogn)$$

No caso médio, o tempo de execução é muito mais próximo do melhor caso do que do pior caso. Para chegarmos a esta conclusão, teremos que compreender como o equilíbrio do particionamento se reflete na recorrência que descreve o tempo de execução. Concluindo que o tempo de execução será igual a equação 2.

2.2 *BucketSort*

BucketSort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes. Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo *BucketSort* recursivamente.

BucketSort funciona do seguinte modo:

- Inicialize um vetor de "baldes", inicialmente vazios.
- Vá para o vetor original, incluindo cada elemento em um balde.
- Ordene todos os baldes não vazios.
- Coloque os elementos dos baldes que não estão vazios no vetor original.

O *BucketSort* tem complexidades de: $O(n^2)$ no pior caso e $O(n + K)$ no melhor e médio caso.

2.3 *HeapSort*

O algoritmo *HeapSort* é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. Foi desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams.

O funcionamento do *HeapSort* pode ser dividido em duas partes.

Na primeira etapa, um heap é criado a partir dos dados. A pilha geralmente é colocada em um vetor com o layout de uma árvore binária completa. A árvore binária completa mapeia a estrutura da árvore binária nos índices do vetor; cada índice do vetor representa um nó; o índice do pai, do ramo filho esquerdo ou do ramo direito do nó são expressões simples.

Na segunda etapa, um vetor ordenado é criado removendo repetidamente o maior elemento do heap (a raiz do heap) e inserindo-o no vetor. A pilha é atualizada após cada remoção para manter a propriedade da pilha. Depois que todos os objetos foram removidos da pilha, o resultado é um vetor com os valores ordenados.

Embora um pouco mais lento na prática na maioria das máquinas do que um *QuickSort* bem implementado, ele tem a vantagem de um tempo de execução O(n log n) para todos os casos.

3. Desenvolvimento

Os algoritmos foram implementados na linguagem Python. Os testes foram feitos em uma máquina com as seguintes configurações: sistema operacional Linux Mint 19.04 com Kernel x86_64 Linux 5.0.0-31-generic; processador Intel Core i7-8550U, com 8 núcleos, memória RAM de 8GB e GPU GeForce 930MX 2GB.

Os testes foram realizados com entradas de números de 5000, 10000, 20000, para cada caso de teste, melhor caso, caso médio e pior caso. Os algoritmos foram executados três vezes para cada entrada de número específico e caso de teste, depois foi retirada a média e gerado três gráficos, contendo o tempo de execução para cada entrada de número e caso de teste.

3.1 Melhor Caso

A figura 1 mostra o resultado com o melhor caso, ou seja, a entrada de números é ordenada. Podemos observar que o *HeapSort* teve o menor tempo, o *BucketSort* teve o segundo melhor tempo, já o *QuickSort* teve o pior tempo, altamente perceptível.

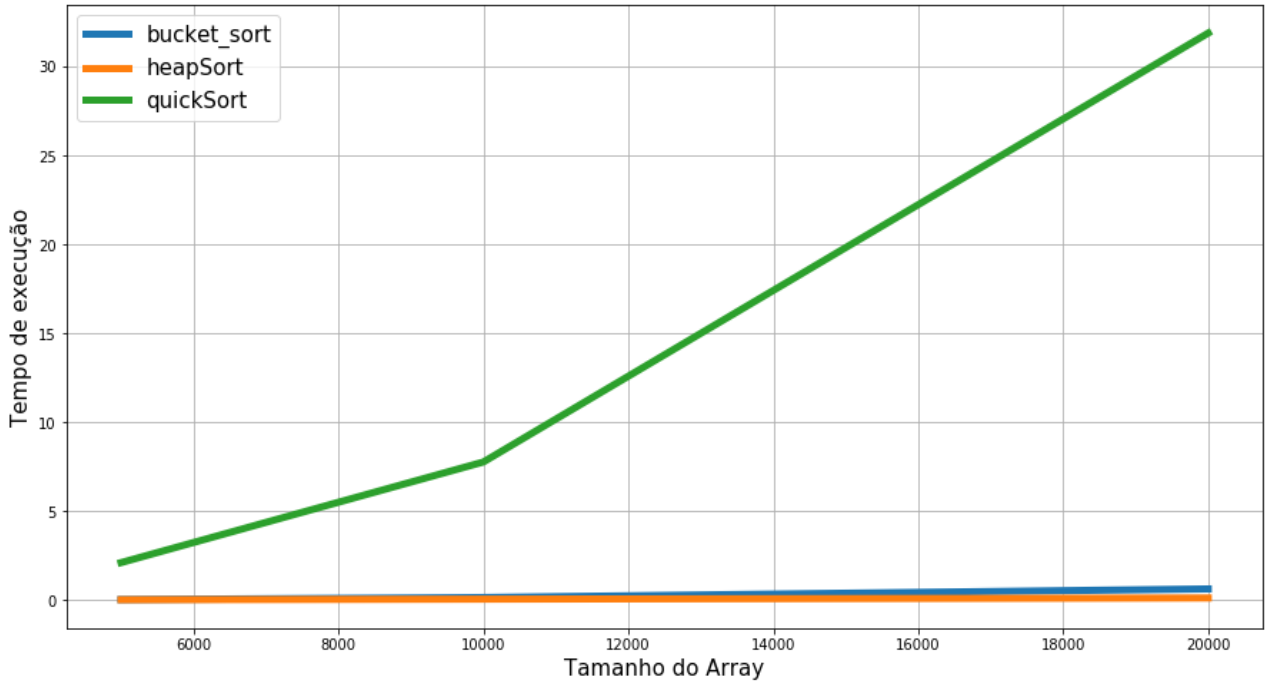


Figura 1: Entrada de array para N valores crescente..

3.2 Caso Médio

A figura 4 mostra o resultado com o caso médio, ou seja, onde a entrada de números é aleatória. Podemos observar que o *BucketSort, foi em disparado o pior algoritmo para resolver esse tipo de problema, enquanto* o *HeapSort* e QuickSort tiveram tempos bem próximos para entrada de números de 1000 até 20000 elementos, ainda assim o *QuickSort* teve o menor tempo nesse caso de teste, tendo um bom tempo de execução.

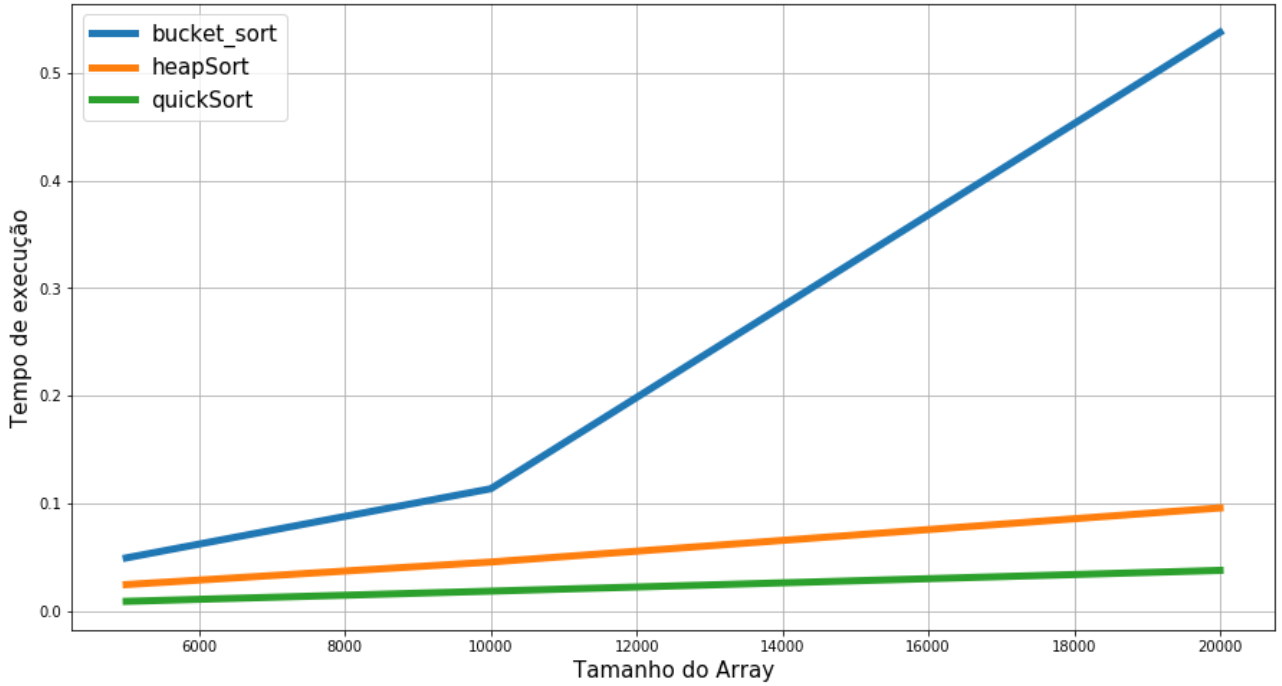


Figura 2: Entrada de array para N valores aleatórios.

3.3 Pior Caso

A figura 5 mostra o resultado do pior caso, ou seja, a entrada de números é de forma decrescente, onde novamente o QuickSort teve problemas para resolver o array de valores ordenados, crescente ou decrescente.os casos

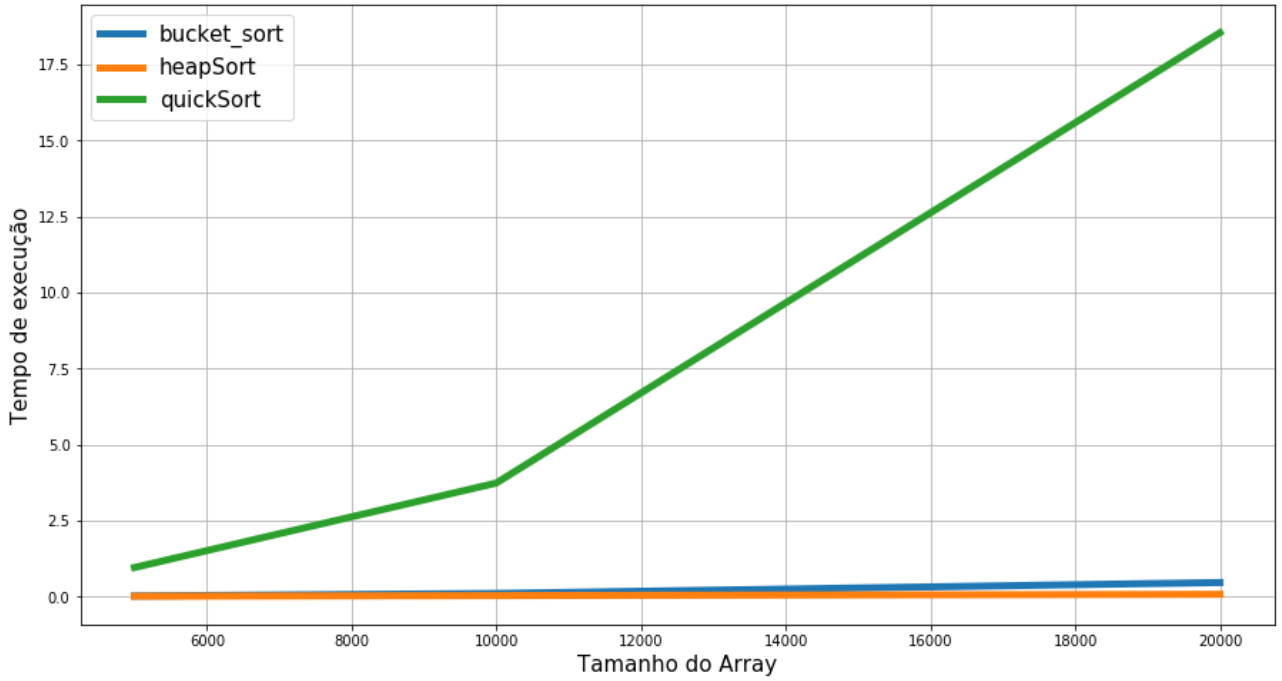
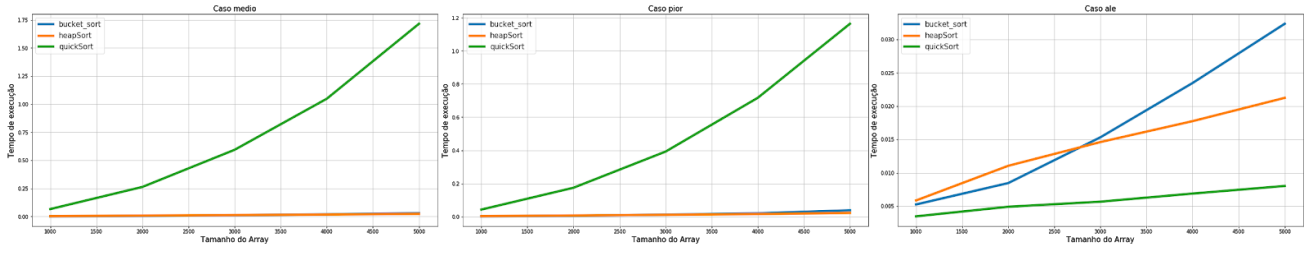


Figura 3: Entrada de array para N valores decrescente.

3.4 Todos os casos em um range menor, valor de N entre 1000 à 5000.

É interessante para ter noção das atenuações que ocorrem no decorrer do tempo para cada tamanho de vetor crescente.



4. Conclusão

Após o estudo comparativo entre os algoritmos apresentados, pode-se concluir que em termos de quantitativos de tempo, o algoritmo *QuickSort* possui um tempo mais elevado de execução em comparação os algoritmos *BucketSort* e *HeapSort*. O *HeapSort* e *BucketSort* possuem tempos de execução similares, no entanto o algoritmo *HeapSort* se sobressai em alguns os casos,

ficando assim perceptível que a utilização de cada algoritmo se dá atraveś do problema que se deve resolver.

Referências

AZEREDO, Paulo A. (1996). *Métodos de Classificação de Dados e Análise de suas Complexidades.* Rio de Janeiro: Campus.

Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2012). *Algoritmos: teoria e prática.* Rio de Janeiro: Elsevier. pp. 145–146–147.

Schaffer, Russel; Sedgewick, Robert (julho de 1993). «The Analysis of Heapsort». *Journal of Algorithms*. 76–100.