Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Análise de Algoritmos de Ordenação

Victor Rodrigues da Silva

29 de outubro de 2021

Resumo

Uma análise dos algoritmos *Quick sort* e *Heap sort*, explicando cenários de pior, melhor caso e caso médio, junto com as suas respectivas complexidades e consumo de memória.

Exibição dos resultados de testes com os 2 algoritmos usando medidas de tempo de execução e considerando margem de variação e tamanho da entrada. Os quais ainda serão comparados com outros 3 métodos comuns de ordenação *Bubble sort, Insertion sort* e *Merge sort*.

Introdução

Os algoritmos que ordenam uma sequência de valores têm grande relevância para o desenvolvimento de inúmeros tipos de sistemas, com necessidades e restrições diferentes. Portanto, saber qual dos vários métodos escolher dependendo da situação é muito importante para qualquer profissional.

Dentre esses algoritmos, existe um grupo deles que é estruturado com base na comparação entre esses valores para deixá-los numa ordem especifica. Nesse relatório serão aprofundados os dois mais eficientes algoritmos desse grupo, o *Quick sort* e *Heap sort.*

Metodologia e Desenvolvimento

Para o desenvolvimento dos algoritmos foi utilizada linguagem C e o compilador GCC da plataforma Unix do WSL(Windows System for Linux), com sistema operacional Ubuntu.

Os gráficos que serão exibidos foram baseados na execução dos algoritmos apresentados e foram coletadas 10 amostras para cada tamanho de entrada (n) para cada caso de ordenação inicial.

Quick Sort

O quick sort segue a filosofia da “divisão e conquista”, que nada mais é que quebrar um problema em problemas menores, ou seja, reexecutar o mesmo código para partes cada vez menores do vetor.

A diferença entre ele e o *Merge sort* que segue a mesma lógica, está centrada principalmente em não gerar cópias do vetor e trocar os valores enquanto faz a divisão ao invés da conquista. O *Quick* divide o vetor em valores menores e maiores que um número escolhido no próprio vetor e chama a si mesmo para resolver esses trechos limitados do vetor.

A escolha de qual valor será o escolhido para esse papel chamado “pivô” é a chave para uma análise mais aprofundada do algoritmo. Observe:

```c
void quickSort(int *vetor, int inicio, int fim)
{
    //condicao de parada
    if (fim - inicio <= 0)
    {
        return;
    }
    int pivo = select_pivo(vetor, inicio, fim) ;
    int i = inicio - 1;
    int j;

    //separa vetor entre menores ou iguais e maiores
    for (j = inicio; j <= fim - 1; j++)
    {
        if (vetor[j] <= pivo)
        {
            i = i + 1;
            swap(vetor, i, j);
        }
    }
    //i indica o ultimo dos menores ou iguais, +1 tornao ultimos dos
maiores
    i++;
    int middle = i;
    //percorre os menores ou iguais separando entre menores e iguais
    for (j = inicio; j < i; j++)
    {
        if (vetor[j] == pivo)
        {
            i = i - 1;
            if (j < fim)
            {
                swap(vetor, i, j);
            }

        }
    }
    // i terminara na posicao
    int equals_start = i;
    // colocando o pivo na primeira posicao dos maiores
    swap(vetor, middle, fim);

    quickSort(vetor, inicio, equals_start -1);
    quickSort(vetor, middle +1, fim);
    return;
}
```

Note que essa é uma variação otimizada do algoritmo que separa o vetor em 3, sendo a parte do meio os valores iguais ao pivô e, portanto, sem necessidade de nova ordenação.

O pior caso do *Quick* tem várias formas de acontecer, uma das mais comuns é quando o pivô escolhido é o primeiro valor do vetor e o vetor já esteja ordenado, nesse caso a divisão será sempre entre um intervalo vazio e outro com n-1. Ou seja, percorre o vetor n-1 vezes o que significa complexidade $O(n^2)$.

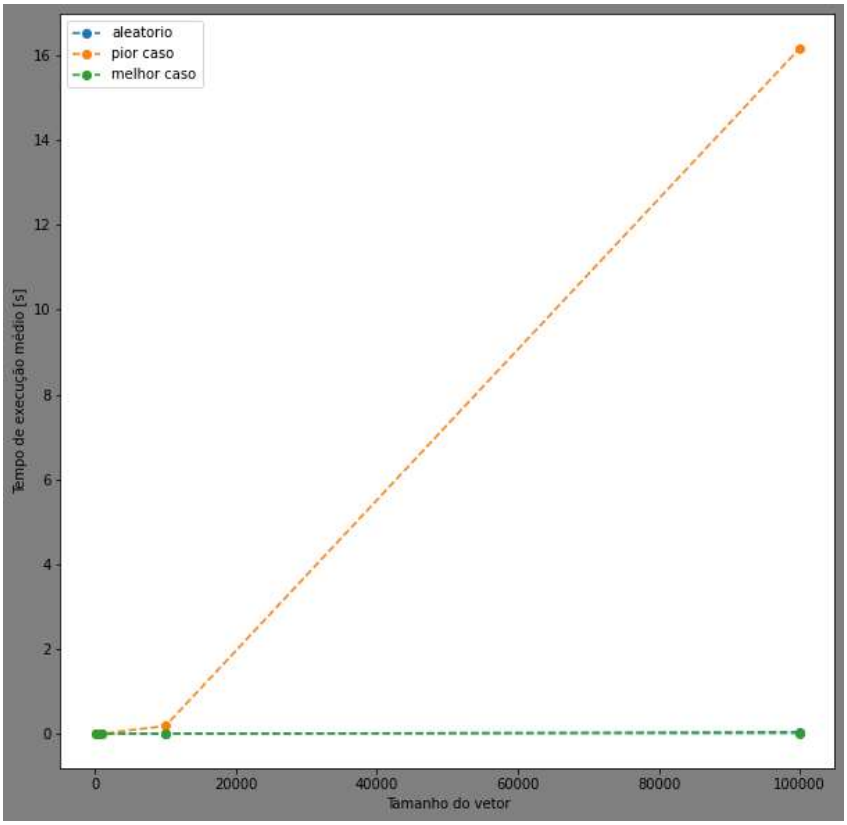
Basicamente o pior ocorre se o pivô escolhido foi sempre o primeiro ou o último número da sequência, isso significa que independente da escolha do pivô o pior caso pode acontecer.

O melhor caso por outro lado, ocorreria sempre que a divisão fosse feita com o maior valor possível, nesse caso, contando na metade e percorrendo o vetor $\log_2(n)$, deixando a sua complexidade $\Omega(n \log(n))$.

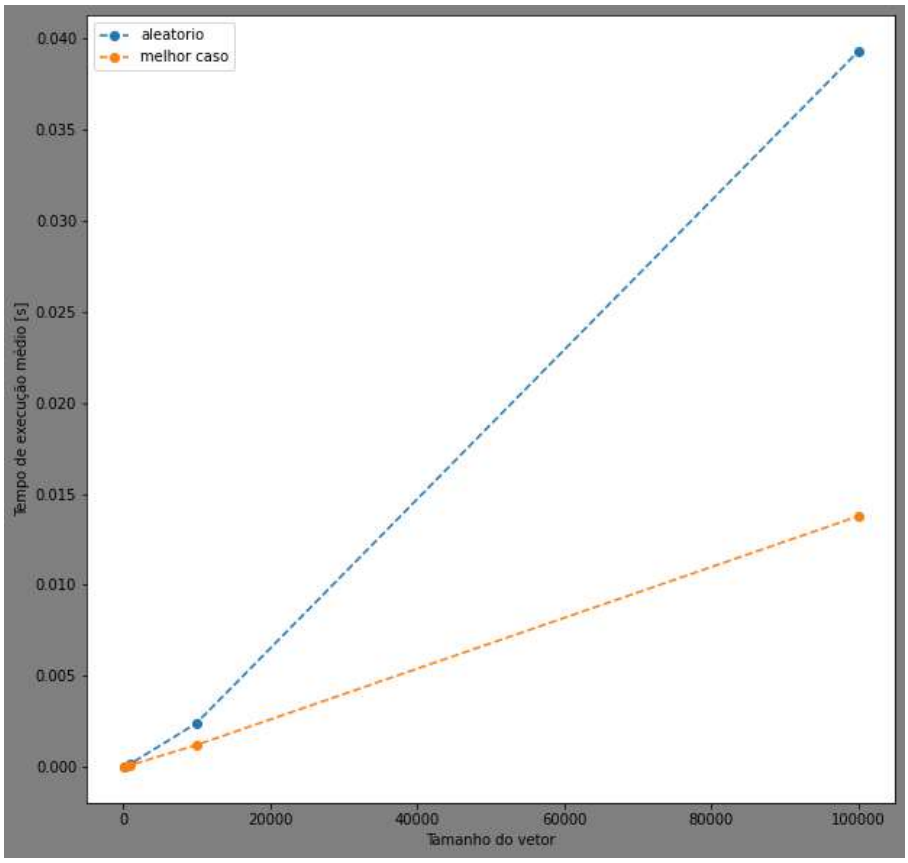
Como o *Quick* é um algoritmo muito instável, é muito difícil criar um vetor que faça exatamente com que cada chamada tenha o seu valor médio como pivô, portanto, foi usado um vetor ordenado, e a escolha do pivô como termo central, assim garantindo por um tempo que as trocas sejam minimizadas.

Quanto ao caso médio, basta considerar a chance de cair no pior caso dado o tamanho de n. Por exemplo, com 1000 valores, a chance de tirar o último ou o primeiro valor é geralmente 0,2%, portanto, maioria das vezes o vetor será dividido em parcelas não nulas, deixando $\Theta(n \log(n))$.

O gráfico exibe a dimensão da diferença entre cada caso:



Agora uma sem considerar o pior caso para facilitar a visualização da diferença entre o caso médio (aleatório) e o melhor caso (pivô mediana e vetor ordenado):



Heap Sort

A estratégia do *Heap sort* está numa estrutura auxiliar em forma de árvore binaria, cujo um valor i do vetor é sempre pai dos valores 2i e 2i+1.

O que o algoritmo faz é basicamente ordenar o vetor de acordo com essa estrutura, tendo em vista que o pai será sempre maior que os filhos, assim, o primeiro valor do vetor será sempre o pai de todos e, portanto, o maior, que será posicionado ao fim e a estrutura de árvore recriada sem o último valor para a próxima iteração.

Aqui está o algoritmo usado, a primeira função é para a construção da árvore e o segundo para a ordenação propriamente:

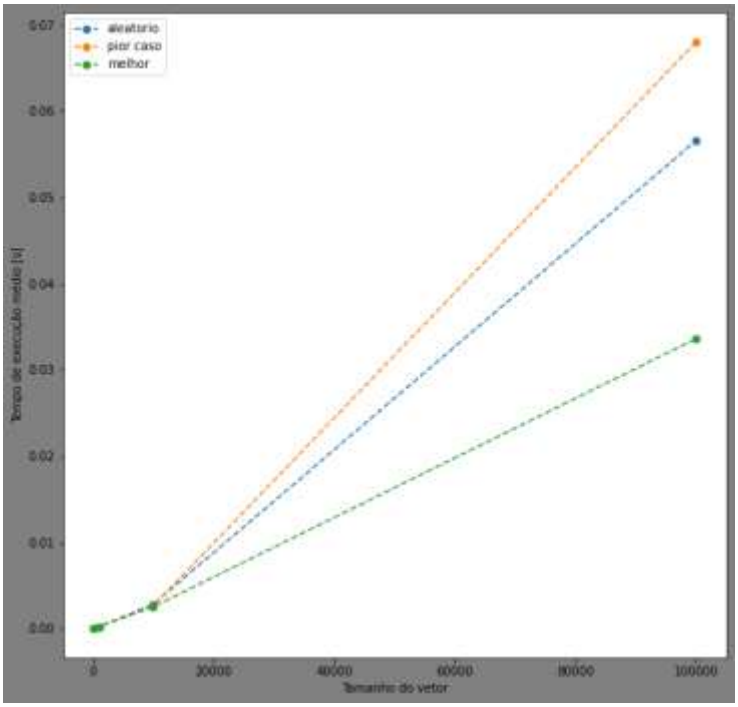
```
void heapifyMax(int *vetor, int pai, int tamanho)
{
    int filho = pai * 2;

    if (filho > tamanho)
        return;

    if (vetor[filho] > vetor[pai] || (filho + 1 <= tamanho && vetor[filho
+ 1] > vetor[pai]))
    {
        if (filho + 1 <= tamanho && vetor[filho + 1] > vetor[filho])
            filho = filho + 1;

        int aux = vetor[pai];
        vetor[pai] = vetor[filho];
```

```
        vetor[filho] = aux;

        heapifyMax(vetor, filho, tamanho);
    }
}

void heapsort(int *vetor, int tamanho)
{
    int ultimoPai = (int)tamanho / 2.0;
    int i;
    for (i = ultimoPai; i >= 1; i--)
        heapifyMax(vetor, i, tamanho);


    // processo de ordena??o
    while (tamanho >= 2)
    {
        // seleciona maior
        int maior = vetor[1];
        vetor[1] = vetor[tamanho];
        vetor[tamanho] = maior;

        tamanho--;
        heapifyMax(vetor, 1, tamanho);
    }
}
```

O pior e melhor caso do *Heap sort* somente dizem respeito ao total de trocas para a construção e renovação, portanto, quão mais próximo o vetor estiver de maxheap (pai maior que filhos) mais eficiente o algoritmo e mais próximo de minheap (filhos maiores que o pai) pior a eficiência.

Ao analisar o algoritmo é possível ver que as trocas e renovações serão, independente da distância para maxheap, n log(n), portanto, tanto O(n log(n)) quanto Ω(n log(n)). Se ambos são iguais o Θ(n log(n)) é igual.

O gráfico mostra quão próximos são os resultados para melhor e pior caso:

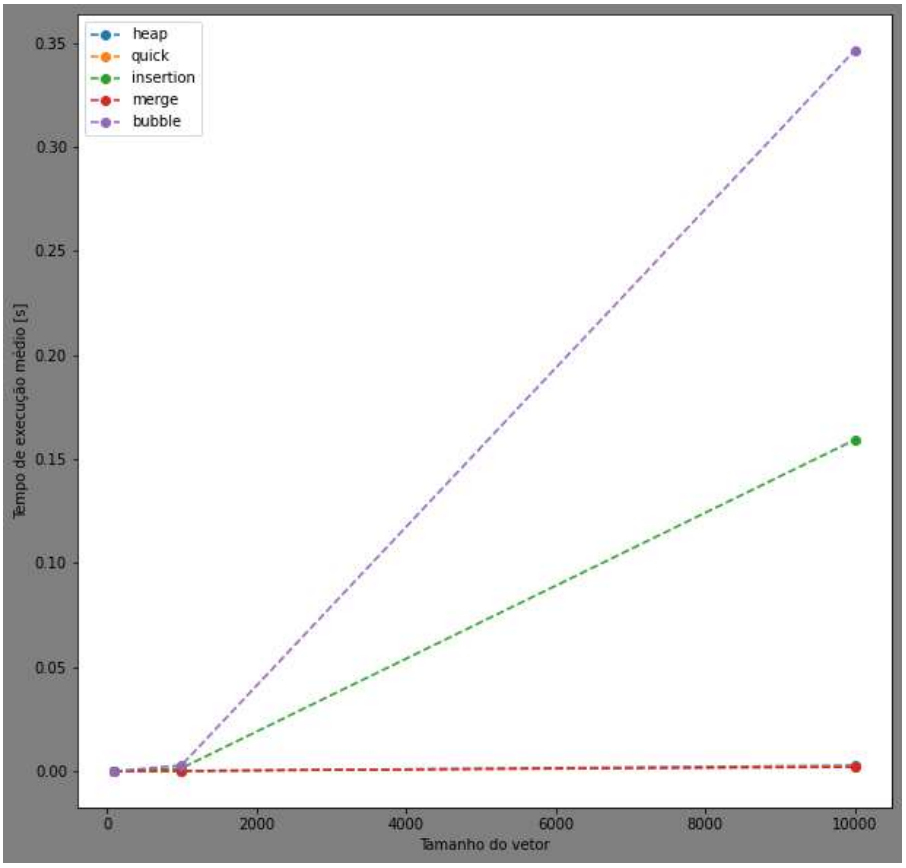


Conclusão

Dentre os algoritmos de ordenação com base na comparação, observa-se que não é possível superar a complexidade $\Omega(n \log(n))$, salvo o *insertion sort*, e ao mesmo tempo alguns algoritmos podem chegar a $O(n^2)$, algo inaceitável.

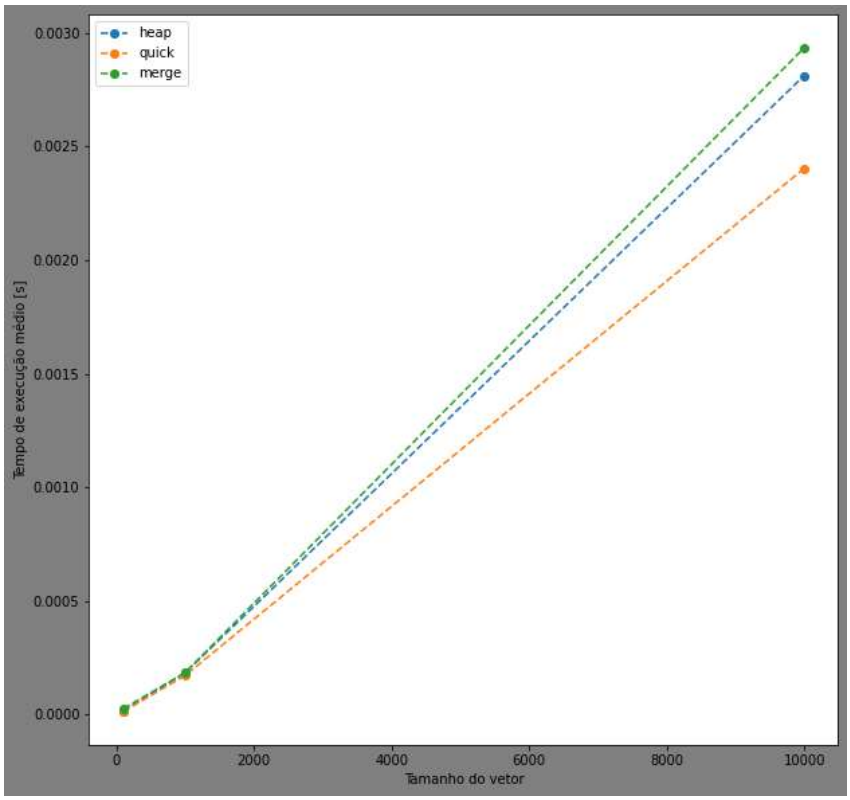
O *insertion sort* por mais que possa chegar a $\Omega(n)$, é extremamente incomum encontrar o cenário para que isso aconteça numa situação prática, ou seja, seu Θ não é n. Assim, cabe escolher entre os algoritmos com $\Theta(n \log(n))$.

O gráfico mostra os resultados do desempenho em tempo de cada um dos métodos de ordenação:



É possível ver quais métodos são os mais eficazes: *merge*, *quick* e *heap*, mas dentre esses, qual é o melhor?

Aprofundando o gráfico e eliminando o *bubble* e *insertion* temos o seguinte:



Merge e *heap* têm tempos muito parecidos, mas ao contemplar a complexidade de memória, lembramos que o *merge* tem complexidade de memória O(n log(n)), um gasto totalmente desnecessário visto sua semelhança em tempo de execução com o *heap sort*.

Já o *quick sort* apresenta o melhor tempo em todas as ocorrências, contudo, o *quick* é assombrado pela possibilidade de acontecer o seu pior caso, um n^2.

Como mostrado no relatório, a simples mudança de seu pivô pode reduzir consideravelmente a chance de que isso aconteça. E como prova o gráfico, na prática, o Θ do *quick* é n log(n) e tem o tempo mais curto entre os demais, portanto, o *quick* pode ser dito como o algoritmo de ordenação mais eficiente dentre o grupo baseado em comparações.

Referências

HEAPSORT. [*S. l.*], 25 set. 2018. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html. Acesso em: 20 nov. 2021.