**Universidade de São Paulo**

**Instituto de Ciências Matemáticas e Computação**

# Análise de algoritmos de ordenação: Quick Sort e Heap Sort

Ana Lívia Ruegger Saldanha

24 de novembro de 2021

**Resumo**

Este relatório apresenta uma análise teórica e experimental acerca da eficiência de dois algoritmos de ordenação: quick sort e heap sort. Com base na bibliografia, cada algoritmo é brevemente debatido a partir de seu funcionamento e implementação em linguagem C, identificando-se suas respectivas funções de eficiência e complexidades assintóticas (notações Big-O e Big-Omega). Essa análise é complementada com os dados obtidos experimentalmente através de medições temporais feitas também em linguagem C. Na parte experimental, os dois algoritmos também são comparados com outros três: bubble sort, insertion sort e merge sort.

---

## 1 Introdução

O problema da ordenação apresentado por Cormen no primeiro capítulo de *Algoritmos: teoria e prática* [1] é frequentemente estudado como introdução aos conceitos de algoritmos e eficiência. Pode ser definido formalmente da seguinte maneira:

- **Entrada:** Uma sequência de $n$ números $< a_1, a_2, ..., a_n >$.
- **Saída:** Uma permutação (reordenação) $< a'_1, a'_2, ..., a'_n >$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Para solucioná-lo, podemos utilizar diversos algoritmos, mas, mesmo que todos sejam corretos e resolvam o problema dado, esses algoritmos podem ser muito diferentes em termos de eficiência. Neste relatório, temos por objetivo debater essa questão através da análise e comparação de cinco algoritmos de ordenação: bubble sort (ordenação por flutuação), insertion sort (ordenação por inserção), merge sort (ordenação por intercalação), quick sort e heap sort, com enfoque nestes dois últimos, sobre os quais apresentaremos uma análise teórica mais aprofundada, visto que os outros já foram estudados no relatório anterior[2].

Primeiramente, buscamos contextualizar esses dois algoritmos (quick sort e heap sort), apresentando seu funcionamento básico e implementações em linguagem C, assim como as funções obtidas através da contagem de operações de cada algoritmo e respectivas análises utilizando o conjunto de notações assintóticas Big-O ($\mathcal{O}$) e Big-Omega ($\Omega$).

Em seguida, apresentamos os resultados de um experimento realizado com os cinco algoritmos através de medições temporais. O experimento tem por objetivo comprovar as discussões levantadas anteriormente para que possamos, por fim, realizar uma análise abrangente desses algoritmos em diferentes cenários.

## 2 Metodologia e desenvolvimento

Abaixo, discutiremos mais a fundo o Quick Sort e o Heap Sort, trazendo também um resumo do que já foi apresentado sobre o Bubble Sort, o Insertion Sort e o Merge Sort [2].

### 2.1 Quick Sort

O *quicksort* é um algoritmo de ordenação muito rápido e eficiente e, portanto, muito utilizado. Sua implementação tradicional, no entanto, apresenta alguns casos patológicos, nos quais o tempo de execução pode ser bastante elevado se comparado ao caso médio. É possível realizar otimizações no algoritmo a fim de evitar os casos patológicos, mas aqui abordaremos o *quicksort* sem otimizações.

Assim como o *mergesort*, o *quicksort* utiliza uma estratégia de divisão e conquista para executar a ordenação. No caso do *quicksort*, no entanto, a partição é feita em torno de um elemento denominado pivô. Existem diversas estratégias para a escolha do pivô; vamos trabalhar com a mais simples delas: quando o pivô é sempre o primeiro elemento.

A seguir, apresentamos a implementação do *quicksort* em linguagem C.[1]

```c
void quicksort(int* v, int ini, int fim)
{
    if (fim <= ini) return;

    int p = ini;
    int i = ini + 1;
    int j = fim;

    while (i <= j)
    {
        while (i <= fim && v[i] <= v[p]) i++;
        while (v[j] > v[p]) j--;

        if (j > i)
        {
            int temp = v[i];
            v[i] = v[j];
            v[j] = temp;
        }
    }

    p = j;
    int temp = v[p];
    v[p] = v[ini];
    v[ini] = temp;
```

[1] As implementações utilizadas no experimento e reproduzidas neste relatório estão de acordo com o material apresentado em aula pelo professor Moacir Ponti[3]. Já as equações e análises assintóticas apresentadas foram baseadas tanto nas aulas quanto na bibliografia, especialmente no livro *Algoritmos: teoria e prática* de Thomas Cormen [1].

```
26
27      quicksort(v, ini, p - 1);
28      quicksort(v, p + 1, fim);
29  }
```

Pela lógica mostrada no código acima, podemos observar que, apesar de o *quicksort* usar uma estratégia de divisão e conquista, seu processo de divisão não é determinístico, ou seja, ele depende do pivô, que terá um valor diferente para vetores diferentes. Adiante, veremos que a eficiência do *quicksort* será definida pela escolha do pivô, e que escolher valores mais centrais resulta em um algoritmo mais eficiente.

O **pior caso** de ordenação ocorre quando a escolha do pivô resulta em uma rotina de particionamento que produz um subproblema com $n-1$ elementos; essa situação se apresenta quando o arranjo de entrada já está completamente ordenado (e poderia ser resolvida pelo *insertion sort* em tempo $\mathcal{O}(n)$). [1] Nesse caso do *quicksort*, podemos notar que a equação de recorrência se assemelha muito à do *bubblesort*, que é da seguinte forma (sendo $c$ o custo das operações de comparação):

$$f(n) = c \cdot (n-1) + f(n-1)$$

Da mesma forma, o número de operações de comparação feitas pelo *quicksort* em função do tamanho $n$ da entrada é dado por:

$$\sum_{k=1}^{n-1} n - k \quad \Longrightarrow \quad f(n) = \frac{n^2 - n}{2}$$

Como $g(n) = n^2$ limita $f(n)$ superiormente para todo $n \geq 1$, podemos afirmar que o *quicksort* pertence a $\mathcal{O}(n^2)$.

No **melhor caso**, em que a divisão é feita da maneira mais equitativa possível e o pivô acaba posicionado no centro, a partição produz dois subproblemas, cada um de tamanho máximo $n/2$ (um é $n/2$, o outro $n/2+1$). Nesse caso, a execução do *quicksort* é muito mais rápida, e sua equação de recorrência é bastante parecida com a do *mergesort*, assumindo a seguinte forma (sendo $c$ o custo das operações de comparação):

$$f(n) = 2f(\frac{n}{2}) + c \cdot n$$

Para a $k$-ésima chamada recursiva, temos:

$$f(n) = k \cdot c \cdot n + 2^k f(\frac{n}{2^k})$$

Considerando o fim da recursão no caso base, temos $k = log_2 n$. Dessa maneira, chegamos à forma fechada da equação de recorrência para o *quicksort*:

$$f(n) = c \cdot n \cdot log_2 n + n$$

Assumindo que as operações constantes na execução do algoritmo tem custo $c = 1$, temos que $g(n) = n \cdot log_2 n$ limita $f(n)$ inferiormente para todo $n \geq 1$. Dessa forma, temos que o *quicksort* pertence a $\Omega(n \cdot log_2 n)$.

Retomando, portanto, podemos afirmar que o *quicksort* pertence a $\mathcal{O}(n^2)$ e a $\Omega(n \cdot log_2 n)$ e, quanto ao uso de memória, pertence a $\mathcal{O}(1)$.

## 2.2 Heap Sort

O *heapsort* resolve o problema da ordenação utilizando uma estrutura de dados específica: uma *heap*, estrutura em árvore binária que satisfaz a seguinte condição: um nó pai sempre terá valor maior ou igual aos valores de seus filhos (*max heap*); ou um nó pai sempre terá valor menor ou igual aos valores de seus filhos (*min heap*). Na execução do algoritmo, a árvore não é de fato construída com nós, mas apenas representada por uma organização específica de um vetor, na qual os nós filhos tem sempre índice $2i$ (filho da esquerda) e $2i + 1$ (filho da direita), sendo $i$ o índice do nó pai. Devido a essa lógica, a implementação em C desse do *heapsort* é feita de maneira a ignorar o primeiro elemento do vetor (de índice zero).

Seu funcionamento utiliza da estratégia de seleção, como no *selection sort* — este sendo, no entanto, um algoritmo nada eficiente (pertence à classe $\mathcal{O}(n^2)$). No *heapsort*, a estrutura de dados garante uma complexidade de tempo muito inferior ($\mathcal{O}(n\ log\ n)$) à da ordenação por seleção "tradicional".

A implementação do *heapsort* é feita em três funções. A função *heapify* é responsável por garantir que um nó esteja em condição de *heap*. Caso faça uma troca, faz uma chamada recursiva para o filho que sofreu a troca, de forma que garante que a condição de *heap* não seja quebrada (se for quebrada, será corrigida). Abaixo, reproduzimos a implementação da função *heapify*, em linguagem C, para a condição de *max heap*:

```c
void max_heapify(int *v, int p, int N)
{
    int f = p * 2;

    if (f > N) return;

    if (v[f] > v[p] || (f + 1 <= N && v[f + 1] > v[p]))
    {
        if (f + 1 <= N && v[f + 1] > v[f])
            f = f + 1;

        int tmp = v[p];
        v[p] = v[f];
        v[f] = tmp;

        max_heapify(v, f, N);
    }
}
```

A segunda função é responsável pela construção da estrutura *heap* em todo o vetor, chamando a função *heapify* para todos os nós que possuem filhos (todos os nós entre o nó de índice $i = n/2$ e o nó raiz). Segue a reprodução dessa função em linguagem C, chamada *build maxheap*:

```c
void build_maxheap(int *v, int N)
{
    int m = N / 2;
```

```
4
5       for (int p = m; p >= 1; p--)
6       {
7           max_heapify(v, p, N);
8       }
9   }
```

Por fim, temos a função que realiza a etapa de seleção propriamente dita, selecionando sempre o maior valor (o nó raiz da *heap*) e colocando-o no fim da parte desordenada do vetor, repetindo esse processo (reconstrução da *heap* e seleção do maior valor) para a parte desordenada até que o vetor esteja completamente ordenado. A implementação dessa função em linguagem C é dada da seguinte forma:

```
1   void heapsort(int *v, int N)
2   {
3       if (N < 1)
4           return;
5
6       build_maxheap(v, N);
7
8       while (N >= 2)
9       {
10          int maior = v[1];
11          v[1] = v[N];
12          v[N] = maior;
13
14          N--;
15
16          max_heapify(v, 1, N);
17      }
18  }
```

Dada a implementação, a análise de complexidade do *heapsort* precisa considerar as operações executadas pelas três funções. Considerando como operação mais relevante a comparação ($c$), para a função *heapify* temos:

$$h(n) = 6c + h(\frac{n}{2})$$

Desenvolvendo a função, podemos encontrar sua forma genérica pra a $k$-ésima chamada recursiva:

$$h(n) = k \cdot 6c + h(\frac{n}{2^k})$$

Considerando o fim da recursão no caso base, temos $k = log_2 n$. Dessa maneira, chegamos à forma fechada da equação de recorrência da função *heapify*:

$$h(n) = log_2 n \cdot 6c$$

Para a construção da *maxheap*, a função *heapify* é chamada $n/2$ vezes; portanto, temos que a função de eficiência do *heapsort* é da seguinte forma:

$$f(n) = \frac{n}{2} \cdot log_2 n \cdot 6c + g(n)$$

Sendo $g(n)$ a função referente às etapas de seleção e remontagem da *maxheap* (que podem ser observadas na implementação da função *heapsort*), temos:

$$g(n) = n \cdot log_2 n \cdot 6c$$

Somando as duas partes, chegamos, finalmente, a uma função de eficiência para o *heapsort*:

$$f(n) = \frac{3}{2} n \cdot log_2 n \cdot 6c$$

Dada a equação acima, podemos afirmar que o *heapsort* pertence tanto a $\mathcal{O}(n \cdot log_2 n)$ quanto a $\Omega(n \cdot log_2 n)$ e, portanto, também a $\Theta(n \cdot log_2 n)$. Quanto ao uso de memória, o *heapsort* pertence a $\mathcal{O}(1)$.

### 2.3 Comparação com Bubble Sort, Insertion Sort e Merge Sort

Para fins de comparação, a Tabela 1 apresenta as informações essenciais acerca da complexidade do *bubble*, do *insertion* e do *mergesort*, como demonstrado no relatório anterior[2].

| Algoritmo | $\mathcal{O}$ | $\Omega$ | Memória auxiliar | Melhor caso | Pior caso |
|---|---|---|---|---|---|
| Bubble Sort | $\mathcal{O}(n^2)$ | $\Omega(n^2)$ | $\mathcal{O}(1)$ | Vetor ordenado | Vetor inversamente ordenado |
| Insertion sort | $\mathcal{O}(n^2)$ | $\Omega(n)$ | $\mathcal{O}(1)$ | Vetor ordenado | Vetor inversamente ordenado |
| Merge Sort | $\mathcal{O}(n\ logn)$ | $\mathcal{O}(n\ logn)$ | $\mathcal{O}(n)$ | - | - |

Tabela 1: Resumo acerca das complexidades de tempo e espaço dos algoritmos de ordenação Bubble Sort, Insertion Sort e Merge Sort.

## 3 Resultados

A seguir, apresentamos os resultados dos dois experimentos realizados a fim de estudar, na prática, a eficiência dos algoritmos apresentados no item anterior.

### 3.1 Vetores gerados aleatoriamente

O primeiro experimento realizado[2] teve como objetivo analisar o desempenho dos algoritmos *quicksort* e *heapsort* com vetores de entrada cada vez maiores. Destacamos algumas escolhas que fizemos na implementação do experimento:

- Foram gerados vetores com valores aleatórios entre 0 e $2n$, sendo $n$ o tamanho da entrada.
- Para cada tamanho de vetor, realizamos 10 iterações/medições; os valores apresentados a seguir, tanto na Tabela 2 quanto no gráfico (Figura 1), referem-se à média do tempo de execução nessas 10 iterações.

[2]Este teste foi executado no meu computador pessoal.

- Os testes dos dois algoritmos foram realizados em uma única execução do programa, com exatamente os mesmos vetores, gerando um arquivo .csv para cada algoritmo. Os dados contidos nesses arquivos foram utilizados para plotar o gráfico reproduzido a seguir (Figura 1).

| Tamanho da entrada | Tempo de execução médio [s] | |
|---|---|---|
| | **Quick Sort** | **Heap Sort** |
| 25 | 0.000003 | 0.000003 |
| 100 | 0.000009 | 0.000015 |
| 500 | 0.000059 | 0.000088 |
| 1000 | 0.000126 | 0.000208 |
| 2500 | 0.000296 | 0.000590 |
| 5000 | 0.000678 | 0.001168 |
| 7500 | 0.001151 | 0.002148 |
| 10000 | 0.001510 | 0.002609 |
| 15000 | 0.002300 | 0.003999 |
| 20000 | 0.002731 | 0.004659 |
| 25000 | 0.003396 | 0.006063 |
| 30000 | 0.004278 | 0.007200 |
| 35000 | 0.004967 | 0.008834 |
| 40000 | 0.005708 | 0.010005 |
| 45000 | 0.006185 | 0.011241 |
| 50000 | 0.007425 | 0.012831 |
| 55000 | 0.007901 | 0.013972 |
| 60000 | 0.008458 | 0.015794 |
| 65000 | 0.009106 | 0.016506 |
| 70000 | 0.010163 | 0.018348 |
| 75000 | 0.011181 | 0.020101 |
| 80000 | 0.012034 | 0.021834 |
| 85000 | 0.012664 | 0.023353 |
| 90000 | 0.013683 | 0.025459 |
| 95000 | 0.015500 | 0.027366 |
| 100000 | 0.015319 | 0.028721 |

Tabela 2: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

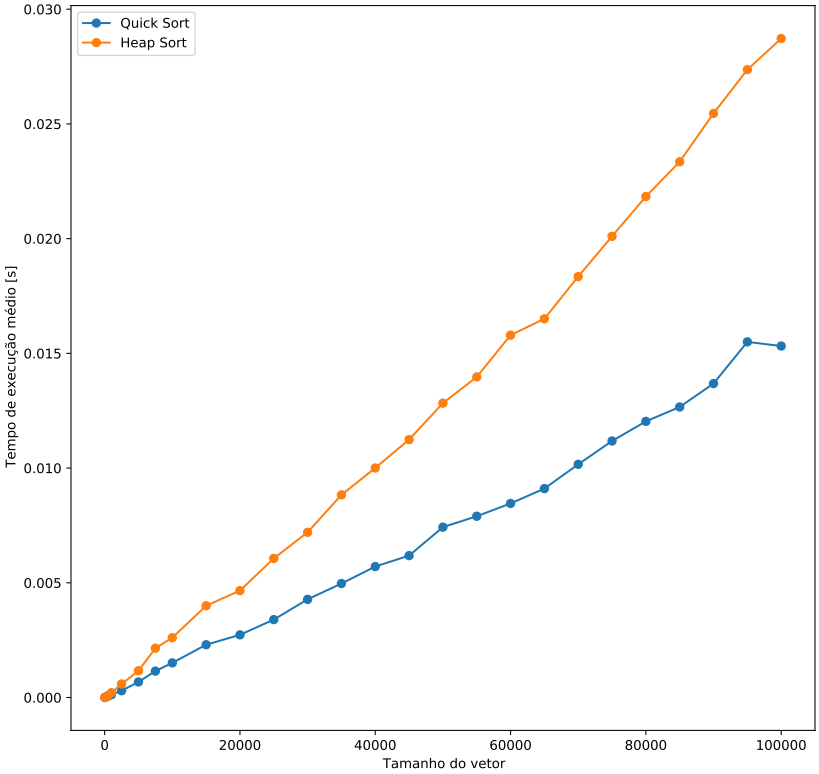


Figura 1: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

Ao observar o gráfico acima, podemos notar que, como esperado, ambos os algoritmos tem comportamento parecido, com caso médio de complexidade de tempo $n \cdot logn$. O *quicksort*, no entanto, mostrou-se significativamente mais rápido em todos os casos.

### 3.2 Comparação com Bubble Sort, Insertion Sort e Merge Sort

O segundo experimento foi realizado seguindo os mesmos critérios de geração aleatória do teste anterior. Desta vez, no entanto, foram testados simultaneamente os cinco algoritmos (Blubble, Insertion, Merge, Quick e Heap Sort) e a ordenação só foi executada para vetores de tamanho até $n = 50000$, devido à lentidão do *bubblesort* na ordenação de vetores tão grandes. Os resultados do teste são apresentados abaixo.

| Tamanho da entrada | Tempo de execução médio [s] | | | | |
|---|---|---|---|---|---|
| | **Bubble Sort** | **Insertion Sort** | **Merge Sort** | **Quick Sort** | **Heap Sort** |
| 25 | 0.000002 | 0.000002 | 0.000003 | 0.000002 | 0.000002 |
| 100 | 0.000029 | 0.000010 | 0.000011 | 0.000007 | 0.000011 |
| 500 | 0.000591 | 0.000208 | 0.000099 | 0.000043 | 0.000093 |
| 1000 | 0.002292 | 0.000790 | 0.000152 | 0.000095 | 0.000159 |
| 2500 | 0.013896 | 0.004848 | 0.000398 | 0.000264 | 0.000438 |
| 5000 | 0.067162 | 0.019546 | 0.000970 | 0.000600 | 0.000991 |
| 7500 | 0.162504 | 0.044578 | 0.001386 | 0.000943 | 0.001560 |
| 10000 | 0.288760 | 0.077497 | 0.001842 | 0.001212 | 0.002088 |
| 15000 | 0.684054 | 0.174053 | 0.002813 | 0.001871 | 0.003275 |
| 20000 | 1.273150 | 0.318072 | 0.003952 | 0.002687 | 0.004660 |
| 25000 | 1.972511 | 0.479805 | 0.004920 | 0.003353 | 0.006163 |
| 30000 | 2.846070 | 0.699651 | 0.006232 | 0.004282 | 0.007359 |
| 35000 | 3.981880 | 0.948351 | 0.007336 | 0.004790 | 0.008591 |
| 40000 | 5.245678 | 1.242266 | 0.008018 | 0.005444 | 0.009695 |
| 45000 | 6.629168 | 1.607736 | 0.009417 | 0.006244 | 0.011173 |
| 50000 | 8.188427 | 1.966531 | 0.010312 | 0.007138 | 0.012641 |

Tabela 3: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor, com vetores gerados aleatoriamente.

Podemos observar que, como esperado pelo que foi demonstrado no item 2 deste relatório, o *bubblesort* apresenta o pior desempenho dentre os cinco algoritmos, seguido pelo *insertion sort*, que apresenta um bom desempenho apenas para vetores de tamanho reduzido ($n \leq 100$). No gráfico a seguir (Figura 2), podemos observar que o *mergesort*, o *quicksort* e o *heapsort* apresentam comportamente bastante similar quando comparados aos outros.

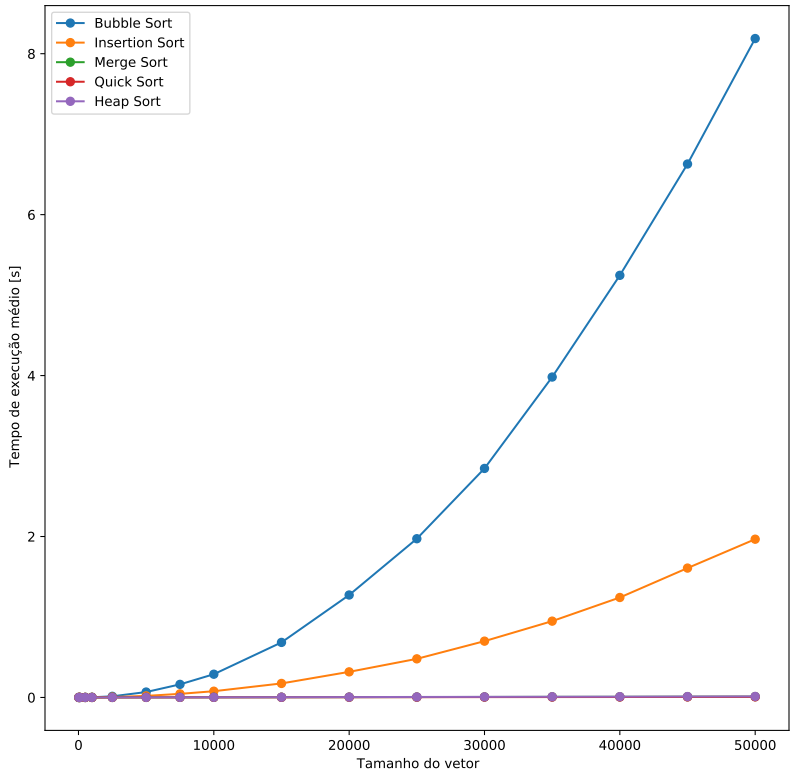


Figura 2: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

Para melhor observar as diferenças entre Merge, Quick e Heap Sort, o gráfico a seguir (Figura 3) compara apenas esses três algoritmos, utilizando os mesmos dados apresentados na Tabela 3 e no gráfico da Figura 2. Podemos notar que, dentre os três, o *quicksort* apresenta o menor tempo médio de execução, seguido pelo *mergesort*; o *heapsort* apresenta o pior desempenho dentre os três.

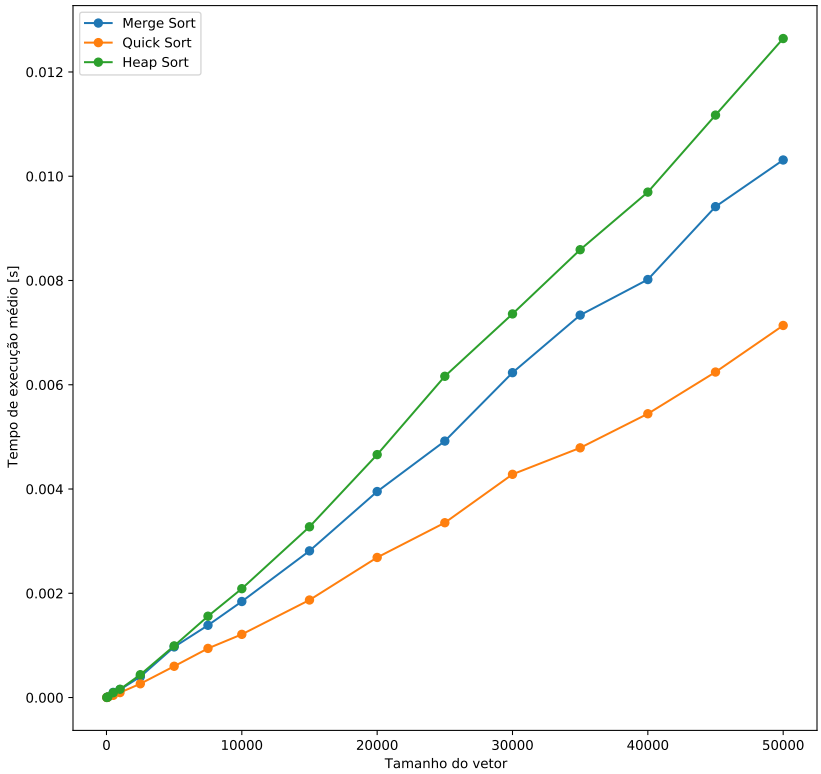


Figura 3: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

### 3.3 Melhor e pior caso

Em um terceiro experimento[3], realizamos testes com vetores de mesmo tamanho ($n = 1000$), porém de diferentes tipos: gerados aleatoriamente, ordenados e inversamente ordenados. De forma complementar aos experimentos anteriores, este teste visou confirmar o que já foi apontado a respeito dos melhores e piores casos para cada algoritmo.

| Tipo de vetor (tamanho $n = 100$) | Tempo de execução médio [s] | | | | |
|---|---|---|---|---|---|
| | **Bubble Sort** | **Insertion Sort** | **Merge Sort** | **Quick Sort** | **Heap Sort** |
| Gerado aleatoriamente | 0.002829 | 0.001202 | 0.000184 | 0.000119 | 0.000188 |
| Ordenado | 0.001492 | 0.000006 | 0.000114 | 0.001255 | 0.000170 |
| Inversamente ordenado | 0.003550 | 0.002418 | 0.000114 | 0.001395 | 0.000163 |

Tabela 4: Resultado das medições temporais para ordenação de diferentes tipos de vetor, todos com o mesmo tamanho; a média de tempo de execução foi tirada a partir de 10 iterações para cada tipo de vetor.

Pelos dados apresentados na tabela acima, podemos confirmar que, de fato, o caso do vetor inversamente ordenado é o pior caso tanto para o *bubblesort* quanto para o *insertion sort*. Por outro lado, o vetor ordenado é o melhor caso

[3]Diferentemente do anterior, este teste foi realizado no GDB Online para evitar as otimizações de branch prediction do meu computador (que melhoram o significativamente o desempenho da ordenação no pior caso).

também para ambos, ainda que, como esperado, a diferença entre o melhor e o pior caso seja muito mais expressiva no *insertion sort*. No caso do *mergesort*, podemos observar que não há diferença considerável entre os tempos de execução para diferentes tipos de vetor[4].

O *quicksort*, como esperado, apresenta o pior desempenho quando testado com vetores ordenados (sejam eles ordenados de maneira crescente ou decrescente). Já o *heapsort*, da mesma maneira que o *merge*, apresenta desempenho similar na ordenação dos três tipos de vetores, não possuindo melhor ou pior caso.

## 4 Conclusão

Através dos resultados dos experimentos, pudemos confirmar as análises apresentadas no Item 2 deste relatório. Retomando rapidamente o que já foi levantado a respeito do Bubble e do Insertion Sort:

- O *bubblesort* é o menos eficiente dos algoritmos em termos de tempo de execução e número de operações executadas; quanto à memória utilizada, apresenta vantagem em relação ao *mergesort* por não utilizar memória auxiliar. No entanto, seu desempenho é muito inferior aos outros e a sua utilização não é recomendada.

- Em termos de tempo de execução e número de operações executadas, o *insertion sort* mostrou-se melhor que o *bubblesort* e pior que os outros três; quanto ao uso de memória, apresenta vantagem em relação ao *mergesort* por não necessitar de memória auxiliar, da mesma forma que os outros quatro algoritmos. Também observa-se que este é um algoritmo bastante eficiente para entradas suficientemente pequenas ($n \leq 100$).

Os outros três algoritmos (*mergesort*, *quicksort* e *heapsort*) possuem ótimo desempenho em termos de tempo de execução e a escolha entre os três deve observar diferenças mais sutis. Em suma:

- O tempo de execução médio do *quicksort* é o menor dentre os três e ele não necessita de memória auxiliar em seu funcionamento. No entanto, possui a desvantagem de apresentar casos patológicos, quando os vetores de entrada já estão ordenados ou majoritariamente ordenados, podendo atingir a complexidade de $\mathcal{O}(n^2)$. Esse problema pode ser mitigado por otimizações na escolha do pivô.

- Dentre os três, aquele que apresenta o segundo melhor tempo de execução é o *mergesort*. Em relação ao *quicksort*, apresenta a vantagem de não encontrar casos patológicos, mas também a desvantagem de necessitar de memória auxiliar para executar a ordenação. Dessa forma, não é recomendada a sua utilização quando há escassez de memória.

- O *heapsort* apresenta o terceiro melhor desempenho dentre os algoritmos estudados. Apesar de ser mais lento que o *quicksort* e o *mergesort*, configura-se como uma opção segura quando se quer evitar tanto os piores casos do *quicksort* quando a utilização de memória auxiliar demandada pelo *mergesort*.

[4] Apesar de o caso do vetor gerado aleatoriamente ter apresentado maior tempo médio de execução, considero que os dados estejam ainda consistentes com o esperado, pois a diferença entre as médias nos três casos é muito pequena.

## Referências


[1] Thomas H. Cormen et al. *Algoritmos: teoria e prática*. Rio de janeiro: Elsevier, 2012.

[2] Ana Lívia Ruegger Saldanha. Análise de algoritmos de ordenação: Bubble Sort, Insertion Sort e Merge Sort. Novembro de 2021.

[3] Moacir Antonelli Ponti. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.

[4] Fernando Pereira dos Santos. Material da disciplina Laboratório de Introdução à Ciência da Computação II (SCC0220). ICMC-USP, 2º semestre de 2021.

[5] Fernando Pereira dos Santos. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.

[6] Sorting algorithm. Disponível em: <https://en.wikipedia.org/wiki/Sorting_algorithm>. Acesso em: 29 de outubro de 2021.