Análise de Desempenho de Algoritmos de Ordenação

Afonso Lucas

Novembro 2024

A análise teórica da complexidade de algoritmos é fundamental para embasar decisões práticas, todavia os resultados teóricos raramente refletem a realidade, influenciada por diversos fatores externos que dificultam a obtenção de resultados ideais. Este trabalho apresenta uma análise experimental da complexidade de tempo de três algoritmos de ordenação amplamente conhecidos por sua eficiência: Merge Sort, Quick Sort e Heap Sort.

Setup de Testes

Para testar os algoritmos, foi implementado um script na linguagem C que realiza a ordenação de conjuntos numéricos gerados aleatoriamente. Os testes foram conduzidos em vetores de tamanhos 10.000, 100.000, 1.000.000, 10.000.000 e 100.000.000 elementos. Cada experimento foi repetido 50 vezes, e foram calculados a média e o desvio padrão dos tempos de execução registrados. A medição do tempo foi realizada utilizando a função `clock()` da biblioteca padrão `<time.h>`. Para a correta execução do script, recomenda-se a utilização de um sistema *Unix-like* para a compilação do código.

Os experimentos foram realizados em um MacBook Pro equipado com processador Apple M1 (8 núcleos), 16 GB de memória unificada e SSD de 1 TB. O sistema operacional utilizado foi o macOS Sonoma 14.7.

Gerando Dados de Teste

Para garantir maior consistência nos resultados, os algoritmos serão testados utilizando os mesmos vetores gerados aleatoriamente. Dessa forma, é necessário gerar as amostras antes de iniciar os testes. Em sistemas *Unix-like*, que geralmente possuem a ferramenta `Make` pré-instalada, é possível gerar os dados de teste executando o seguinte comando no terminal, na pasta onde os arquivos estão localizados (não copiar o cifrão):

```
$ make generate
```

Caso a ferramenta `Make` não esteja instalada, siga os dois passos abaixo. Primeiramente, compile o programa responsável por gerar os dados:

```
$ gcc generate.c -o gen_out
```

Após a compilação, você deve ter um novo arquivo em sua pasta (`gen_out`), execute o programa gerado:

```
$ ./gen_out
```

A execução do programa pode levar de 10 a 20 minutos devido à grande quantidade de dados a ser gerada. Ao final do processo, é esperado que a estrutura de pastas gerada seja a seguinte:

```
- generated
  |- 10_000
  |- 100_000
  |- 1_000_000
  |- 10_000_000
  |- 100_000_000
```

Dentro de cada pasta, haverá 50 amostras de números aleatórios gerados, com os arquivos nomeados no padrão `numbers_x.txt`.

Análise de Desempenho

Com o ambiente de teste preparado, realizamos uma análise prática do desempenho de três algoritmos de ordenação: **Merge Sort**, **Quick Sort** e **Heap Sort**. Os três algoritmos foram avaliados utilizando o mesmo conjunto de dados, permitindo uma comparação direta de suas performances.

Merge Sort

O **Merge Sort** é um algoritmo de ordenação amplamente conhecido que utiliza a estratégia de divisão e conquista para organizar conjuntos enumeráveis. Sua abordagem consiste em dividir repetidamente o conjunto de dados em subconjuntos menores até que cada subconjunto contenha apenas um elemento, tornando a ordenação trivial. Após essa divisão, o maior custo computacional ocorre na etapa de fusão (*Merge*), onde dois subconjuntos já ordenados são combinados para formar um único conjunto ordenado e isto se segue até construirmos um conjunto com todos os termos do conjunto inicial, porém ordenados. A seguir temos uma base do código do **Merge Sort** que já possibilita analisar sua complexidade de tempo:

```
static void __mergesort(unsigned int arr[], const int left, const int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        __mergesort(arr, left, mid);
        __mergesort(arr, mid + 1, right);
        __merge(arr, left, mid, right);
    }
}
```

O algoritmo de **Merge Sort** é implementado utilizando recursão, a formula de recorrência para sua complexidade de tempo é dada por:

$$T(n) = \begin{cases} 1, & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + n, & \text{se } n > 1. \end{cases}$$

Resolvendo essa recorrência pelo *Teorema Mestre* temos que a complexidade de tempo para o **Merge Sort** é de $O(n \log n)$.

Testes Computacionais do Merge Sort

Para executar os testes do algoritmo **Merge Sort**, caso você possua a ferramenta `Make` instalada, basta executar o comando a seguir:

```
$ make mergesort
```

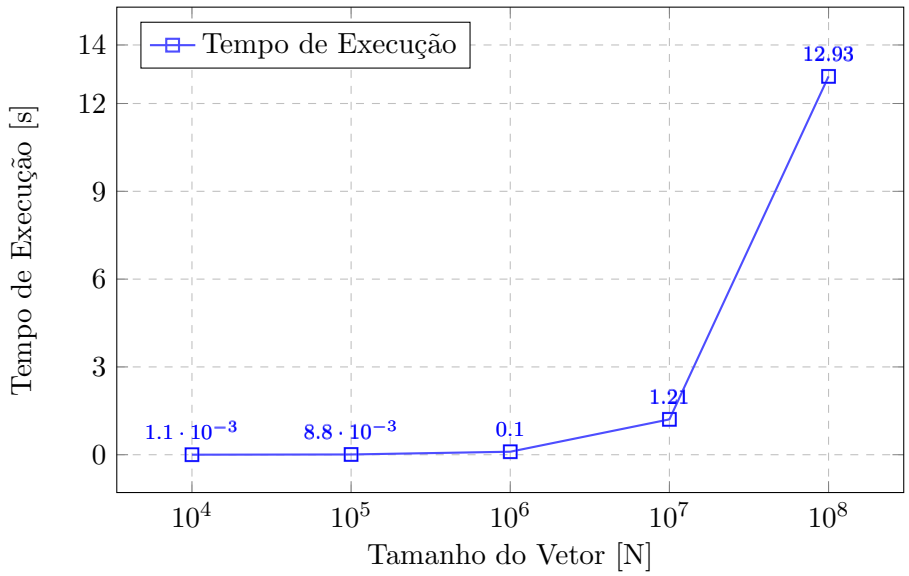
Caso você não tenha o `Make` instalado, primeiro será necessário compilar o código com o seguinte comando:

```
$ gcc main.c ./src/mergesort.c ./src/quicksort.c ./src/heapsort.c -o out -lm
```

Após a compilação, um arquivo executável (`out`) será gerado na pasta atual. Para rodar o programa, basta executar o comando abaixo, passando o nome do algoritmo como argumento na linha de comando:

```
$ ./out mergesort
```

O algoritmo pode levar até 20 minutos para concluir todos os testes. Ao final, teremos os valores das médias e dos desvios padrão para todos os tamanhos de entrada. Após executar o programa no meu ambiente de teste, os seguintes resultados foram obtidos:



Abaixo está uma tabela que apresenta os resultados numéricos dos testes de forma mais clara:

Tamanho de Entrada (N)	**Tempo Médio (s)**	**Desvio Padrão (s)**
10.000	0.0011	0.0006
100.000	0.0088	0.0001
1.000.000	0.1025	0.0036
10.000.000	1.2081	0.0484
100.000.000	12.9262	0.1644

Table 1: Desempenho do Merge Sort

A partir dos resultados experimentais obtidos, é possível observar uma forte semelhança com os resultados analíticos previstos para o algoritmo. Especificamente, ao aumentar o tamanho da entrada em um fator de 10.000×, o tempo de execução do algoritmo não apresentou um crescimento superior a

12.000×. Esse comportamento indica que, embora o aumento no tempo de ordenação tenha sido mais acentuado do que o crescimento linear ($O(n)$), ele se manteve dentro dos limites esperados para a complexidade assintótica de $O(n \log n)$. Isso confirma que o **Merge Sort**, de fato, apresenta um desempenho compatível com o comportamento teórico para entradas de grandes dimensões.

Quick Sort

O **Quick Sort** é um algoritmo de ordenação eficiente que também utiliza a estratégia de divisão e conquista, mas com uma abordagem diferente da do **Merge Sort**. Sua estratégia envolve o particionamento do conjunto em dois subconjuntos com base na escolha de um elemento pivô, de modo que todos os elementos menores que o pivô fiquem à sua esquerda e todos os elementos maiores fiquem à sua direita. Esse processo é repetido recursivamente nos subconjuntos, até que cada partição tenha apenas um elemento, momento em que a ordenação está concluída. Podemos ter uma boa noção dos passos necessários para o algoritmo ao ver o código de implementação:

```
static void __quicksort(unsigned int arr[], int left, int right) {
    if (left < right) {
        unsigned int p = __partition(arr, left, right);

        __quicksort(arr, left, p-1);
        __quicksort(arr, p+1, right);
    }
}
```

A partir disso, podemos chegar em sua equação de recorrência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + n, & \text{se } n > 1. \end{cases}$$

O maior custo computacional do **Quick Sort** está associado à escolha do pivô e ao particionamento dos elementos. Quando uma boa estratégia para a escolha do pivô é adotada, o algoritmo apresenta um desempenho altamente eficiente, com uma complexidade média de $O(n \log n)$. Sua equação de recorrência é semelhante à do **Merge Sort**, e, de fato, no melhor caso, quando o pivô escolhido resulta em uma divisão equilibrada do vetor, o comportamento do **Quick Sort** se aproxima do comportamento do **Merge Sort**.

Vale ressaltar que a eficiência do **Quick Sort**, em comparação ao **Merge Sort**, está relacionada à sua complexidade de espaço, já que mesmo possui desempenho semelhante ao do **Merge Sort**, mas com complexidade de espaço $O(1)$.

Testes Computacionais do Quick Sort

Para executar os testes do algoritmo **Quick Sort**, caso você possua a ferramenta `Make` instalada, basta executar o comando a seguir:

```
$ make quicksort
```

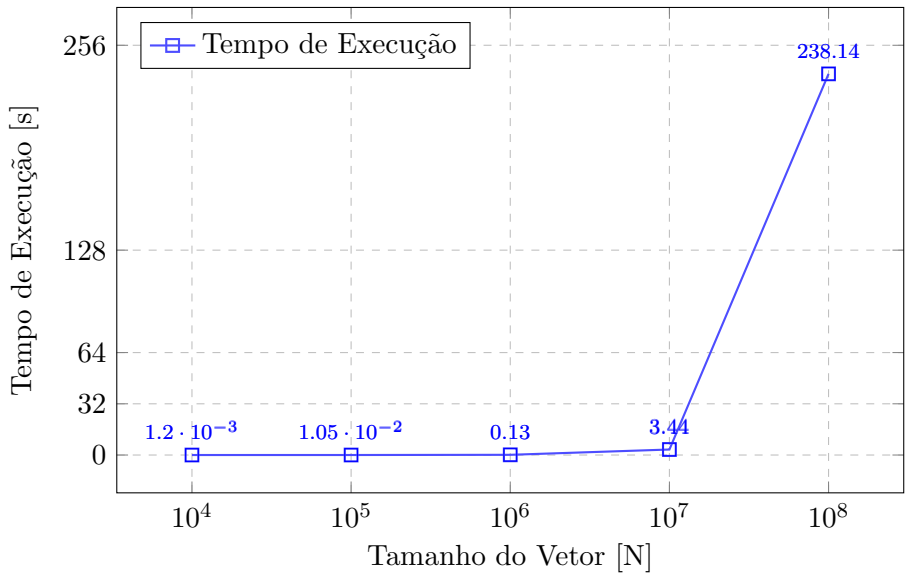
Caso você não tenha o `Make` instalado, é esperado que você já deva ter um arquivo "`out`" na sua pasta atual após os primeiros testes com o **Merge Sort**, mas caso não tenha, será necessário compilar o código com o seguinte comando:

```
$ gcc main.c ./src/mergesort.c ./src/quicksort.c ./src/heapsort.c -o out -lm
```

Após a compilação, um arquivo executável (`out`) será gerado na pasta atual. Para rodar o programa, basta executar o comando abaixo, passando o nome do algoritmo como argumento na linha de comando:

```
$ ./out quicksort
```

O programa pode levar até 4 horas para concluir todos os testes. Ao final, teremos os valores das médias e dos desvios padrão para todos os tamanhos de entrada. Após executar o programa no meu ambiente de teste, os seguintes resultados foram obtidos:



Abaixo está uma tabela que apresenta os resultados numéricos dos testes de forma mais clara:

Tamanho de Entrada (N)	**Tempo Médio (s)**	**Desvio Padrão (s)**
10.000	0.0012	0.0006
100.000	0.0105	0.0001
1.000.000	0.1322	0.0029
10.000.000	3.4416	0.0507
100.000.000	238.1375	1.3521

Table 2: Desempenho do Quick Sort

Baseado nos resultados experimentais obtidos para o **Quick Sort**, é possível identificar que o algoritmo apresenta um crescimento significativo no tempo de execução conforme o tamanho do conjunto de entrada aumenta. Para entradas de tamanho $N = 100.000.000$, o tempo médio atinge 238 segundos. Essa variação reflete a sensibilidade do algoritmo ao tamanho da entrada, especialmente para instâncias maiores. Embora o comportamento médio seja esperado ainda dentro da complexidade $O(n \log n)$, a diferença observada em relação ao **Merge Sort** é clara. O desvio padrão também aumenta de maneira significativa, o que já era esperado, dada a dependência do algoritmo da escolha de um pivô ideal em todas as chamadas para atingir um tempo ideal.

Heap Sort

O **Heap Sort** é um algoritmo de ordenação amplamente conhecido, assim como os outros apresentados até o momento, contudo ele se destaca por alcançar alta eficiência sem recorrer à estratégia de divisão e conquista. O **Heap Sort** utiliza a estrutura de dados **Heap** para realizar a ordenação. Sua abordagem consiste, inicialmente, em construir um **Heap-mínimo** ou **Heap-máximo** com os elementos a serem ordenados, uma operação que possui complexidade $O(n)$ e após a construção do mesmo, os elementos são extraídos, um a um, sempre mantendo as propriedades que definem a estrutura como sendo um **Heap**, até que todos os elementos tenham sido removidos e o vetor esteja completamente ordenado. A operação de extração apresenta uma complexidade de $O(\log n)$.

A implementação é simples, e podemos ver sua divisão em duas etapas de maneira clara no código que se segue:

```c
void hsort(unsigned int arr[], const size_t n) {
    // 1º Construção do heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        __heapify(arr, n, i);
    }
    // 2º Ordenação
    for (unsigned int i = n - 1; i > 0; i--) {
        SWAP(&arr[0], &arr[i]);
        __heapify(arr, i, 0);
    }
}
```

E sua equação de recorrência é a que se segue:

$$T(n) = \begin{cases} 0, & \text{se } n = 1, \\ T(n-1) + \log(n), & \text{se } n > 1. \end{cases}$$

Não iremos entrar em detalhes quanto aos passos para a solução dessa recorrência, mas ao expandi-la obtém-se $T(n) = \log(n!)$. Utilizando a *Aproximação de Stirling*, sabemos que $\log(n!)$ é limitada superiormente por $n\log(n)$, assim conclui-se que a complexidade de tempo do **Heap Sort** é $T(n) = O(n\log(n))$.

Apesar da necessidade de um **Heap** para a ordenação, todo o processo pode ser realizado de forma *in-place* no vetor. Isso ocorre porque o vetor é tratado como uma árvore binária completa, representando o **Heap**, eliminando a necessidade de alocação adicional de memória, assim o algoritmo é eficiente como o **Merge Sort** e o **Quick Sort**, mas mantém uma complexidade de espaço constante $O(1)$.

Testes Computacionais do Heap Sort

Para executar os testes do algoritmo **Heap Sort**, caso você possua a ferramenta `Make` instalada, basta executar o comando a seguir:

```
$ make heapsort
```

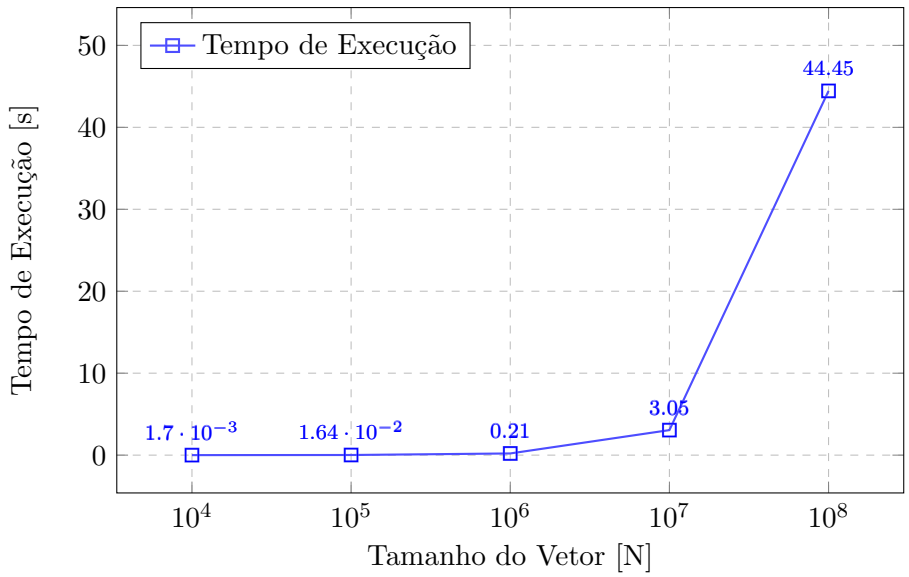
Caso você não tenha o `Make` instalado, é esperado que você já deva ter um arquivo "`out`" na sua pasta atual após os primeiros testes com o **Merge Sort** ou com o **Quick Sort**, mas caso não tenha, será necessário compilar o código com o seguinte comando:

```
$ gcc main.c ./src/mergesort.c ./src/quicksort.c ./src/heapsort.c -o out -lm
```

Após a compilação, um arquivo executável (`out`) será gerado na pasta atual. Para rodar o programa, basta executar o comando abaixo, passando o nome do algoritmo como argumento na linha de comando:

```
$ ./out heapsort
```

O programa pode levar até 1 hora para concluir todos os testes. Ao final, teremos os valores das médias e dos desvios padrão para todos os tamanhos de entrada. Após executar o programa no meu ambiente de teste, os seguintes resultados foram obtidos:



Abaixo está uma tabela que apresenta os resultados numéricos dos testes de forma mais clara:

Tamanho de Entrada (N)	**Tempo Médio (s)**	**Desvio Padrão (s)**
10.000	0.0017	0.0008
100.000	0.0164	0.0001
1.000.000	0.2052	0.0011
10.000.000	3.0533	0.0982
100.000.000	44.4457	0.8842

Table 3: Desempenho do Heap Sort

Com base nos resultados experimentais obtidos, é possível observar resultados consistentes para o **Heap Sort**, ainda refletido a complexidade $O(n \log n)$. Os testes apresentam um resultado até melhor que o **Quick Sort** para quantidades muito grandes, mas ainda não se compara com os resultados obtidos pelo **Merge Sort**.

Conclusão

Com base nos testes, uma aparente conclusão que podemos tomar é que o **Merge Sort** é o algoritmo mais eficiente em relação à complexidade de tempo. Dentro da categoria de algoritmos de divisão e conquista testados, o mesmo se mostrou mais rápido e com resultados mais consistentes do que o **Quick Sort**, todavia essa performance vem com um custo de espaço proporcional ao tamanho da entrada, o que tem maior probabilidade de gerar problemas como um erro de **stack overflow**. O **Quick Sort** normalmente é tido como o algoritmo de ordenação mais rápido do que o **Merge Sort**, mas isto depende muito das otimizações feitas na hora da implementação, a escolha do elemento pivô influencia muito em sua performance e, dependendo da estratégia utilizada, normalmente a mais adotada é a escolha de um pivô aleatório, pode gerar resultados mais inconsistentes, por conta disso já era esperado que tivéssemos o maior valor de desvio padrão, mesmo que não estejamos adotando a escolha de um pivô aleatório, então por mais que seja um algoritmo relativamente eficiente, ainda vale lembrar que seu pior caso ainda se enquadra em $O(n^2)$. Com relação ao **Heap Sort** temos um bom equilíbrio entre eficiência de tempo, onde seu pior caso se encontra na mesma posição do **Merge Sort**, sendo $O(n \log n)$, e sem a necessidade de alocação de memória adicional, sendo uma boa alternativa com relação à eficiência em um algoritmo que não usa a estratégia de divisão e conquista.

References

[Feo19a] Paulo Feofiloff. *Heap Sort - Aula sobre Heap Sort*. Atualizado em 2019-02-01. © Paulo Feofiloff, DCC-IME-USP. 2019. URL: `https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html`.

[Feo19b] Paulo Feofiloff. *Merge Sort - Aula sobre Merge Sort*. Atualizado em 2019-03-01. © Paulo Feofiloff, DCC-IME-USP. 2019. URL: `https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html`.

[Feo19c] Paulo Feofiloff. *Quick Sort - Aula sobre Quick Sort*. Atualizado em 2019-03-01. © Paulo Feofiloff, DCC-IME-USP. 2019. URL: `https://www.ime.usp.br/~pf/algoritmos/aulas/quick.html`.

[Indnd] Indian Institute of Information Technology Design and Manufacturing. *COM 501 Advanced Data Structures and Algorithms - Lecture Notes: Recurrence Relations*. Chennai, India. n.d.