Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Pós-graduação em Ciência da Computação

Análise de Algoritmos de Ordenação

Diego Martos Buoro 21201931106
Danilo dos Santos Bezerra 21201831086

29 de novembro de 2019

Sumário

1 Introdução **5**

2 Algoritmos **5**

2.1 Insertion Sort 5

2.1.1 Análise de Complexidade 8

2.2 Selection Sort 8

2.2.1 Análise de Complexidade 10

2.3 Bubble Sort 11

2.3.1 Análise de Complexidade 12

2.4 Merge Sort 13

2.4.1 Análise de Complexidade 16

2.5 Quick Sort 16

2.5.1 Análise de Complexidade 17

2.6 Heap Sort 18

2.6.1 Análise de Complexidade 20

3 Quadro de Comparação **20**

4 Ambiente de execução **22**

5 Resultados e análise **22**

5.1 Resultados 22

5.2 Análise 24

6 Conclusão **28**

7 Referências **29**

Lista de Figuras

1 Situação do elemento x antes de sua inserção no subvetor ordenado (começo do laço). 6

2 Situação do elemento x depois de sua inserção no subvetor ordenado (final do laço). 6

3 Ordenação de 5 cartas de baralho pelo Insertion Sort, mostrando, de cima para baixo, a disposição dos dados a cada interação 7

4 Ordenação de 5 cartas de baralho pelo Selection Sort, mostrando, de cima para baixo, a disposição dos dados a cada iteração 10

5 Ordenação de 5 cartas de baralho pelo Bubble Sort, mostrando, de cima para baixo, a disposição dos dados a cada iteração 12

6 Ordenação de um vetor pelo Mergesort. As setas vermelhas representam a divisão em subproblemas que é realizada pelas chamadas recursivas. Uma vez que o subproblema possa ser resolvido diretamente (elementos em cinza), os elementos são recombinados e os subvetores reconstruídos pelo procedimento *Merge*, como indicam as setas verdes, até obtermos o vetor ordenado. 14

7 Gráfico da complexidade dos algoritmos estudados com entrada aleatória. 25

8 Gráfico da complexidade dos algoritmos estudados com entrada ordenada. 26

9 Gráfico da complexidade dos algoritmos estudados com entrada inversa. 27

10 Gráfico da complexidade dos algoritmos linearítmicos estudados para entradas aleatória. 28

Lista de Tabelas

1 Complexidade do algoritmo em função do tamanho de entrada n e da disposição de dados. 21

2 Disposição de dados para cada algoritmo no pior (inverso) ou melhor caso (ordenado). 21

3 Comparação de tempo do Insertion Sort em segundos 22

4 Comparação de tempo do Bubble Sort em segundos 23

5 Comparação de tempo do Selection Sort em segundos 23

6 Comparação de tempo do Merge Sort em segundos 23

7 Comparação de tempo do Quick Sort em segundos 24

8 Comparação de tempo do Heap Sort em segundos 24

1 Introdução

Entre os problemas mais estudados desde o surgimento da Ciência da Computação, está a ordenação de um conjunto de elementos seguindo um critério. Com o surgimento de algoritmos e diferentes estratégias para a ordenação de elementos, criou-se a necessidade de fazer uma análise mais aprofundanda da performance desse conjunto de operações. Afinal, o tempo necessário para a execução de um algoritmo é crucial em aplicações que requeiram um tempo curto de execução. O desenvolvimento de técnicas de análise de algoritmos permitiu a realização dessa análise de forma rigorosa e científica.

Este trabalho se propõe a realizar, em um ambiente de execução definido, uma análise da complexidade de diferentes algoritmos de ordenação por comparação, por meio de testes empíricos ou, em outras palavras, fazer uma análise do tempo em função do tamanho e da disposição do conjunto de dados. Tanto este relatório como os códigos que foram feitos para a execução do teste podem ser encontrados aqui ou na seção "Referências".

2 Algoritmos

Fundamentalmente, os algoritmos, a seguir, buscam resolver o seguinte problema: dado um vetor de tamanho n e $v[0..n-1]$, rerranja-se o vetor de tal modo que fique ordenado, ou seja, que possui a seguinte propriedade: $v[0] \leq v[1] \leq ... \leq v[n-2] \leq v[n-1]$. Durante explicação dos algoritmos, os pseucódigos foram escritos levando em consideração as características pecualiares da linguagem de programação C.

2.1 Insertion Sort

O *insertion sort*, também chamado por ordenação por inserção, é um algoritmo onde dado um vetor $v[0..n-1]$, e sendo o acesso dos elementos da esquerda para a direita, coloca, na iteração i, o elemento $v[i]$ - chamado de x, como mostra a figura 1 - no vetor ordenado $v[0..i-1]$. No final do laço da iteração i, temos que v[0..i] está ordenardo, situação representada pela figura 2.

Figura 1: Situação do elemento x antes de sua inserção no subvetor ordenado (começo do laço).

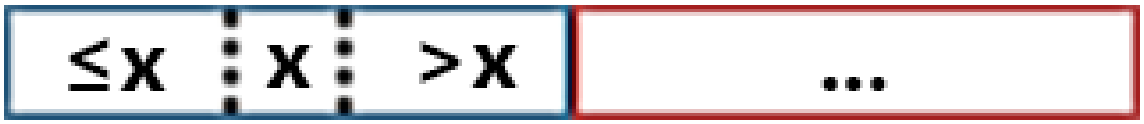

Figura 2: Situação do elemento x depois de sua inserção no subvetor ordenado (final do laço).

Assim, o segundo laço (da linha 4 até 7) é considerado a operação elementar desse algoritmo, realizando o trabalho de ordenação. A figura 3 mostra um exemplo de ordenação O pseudocódigo 1 abaixo representa algoritmo descrito anteriormente.

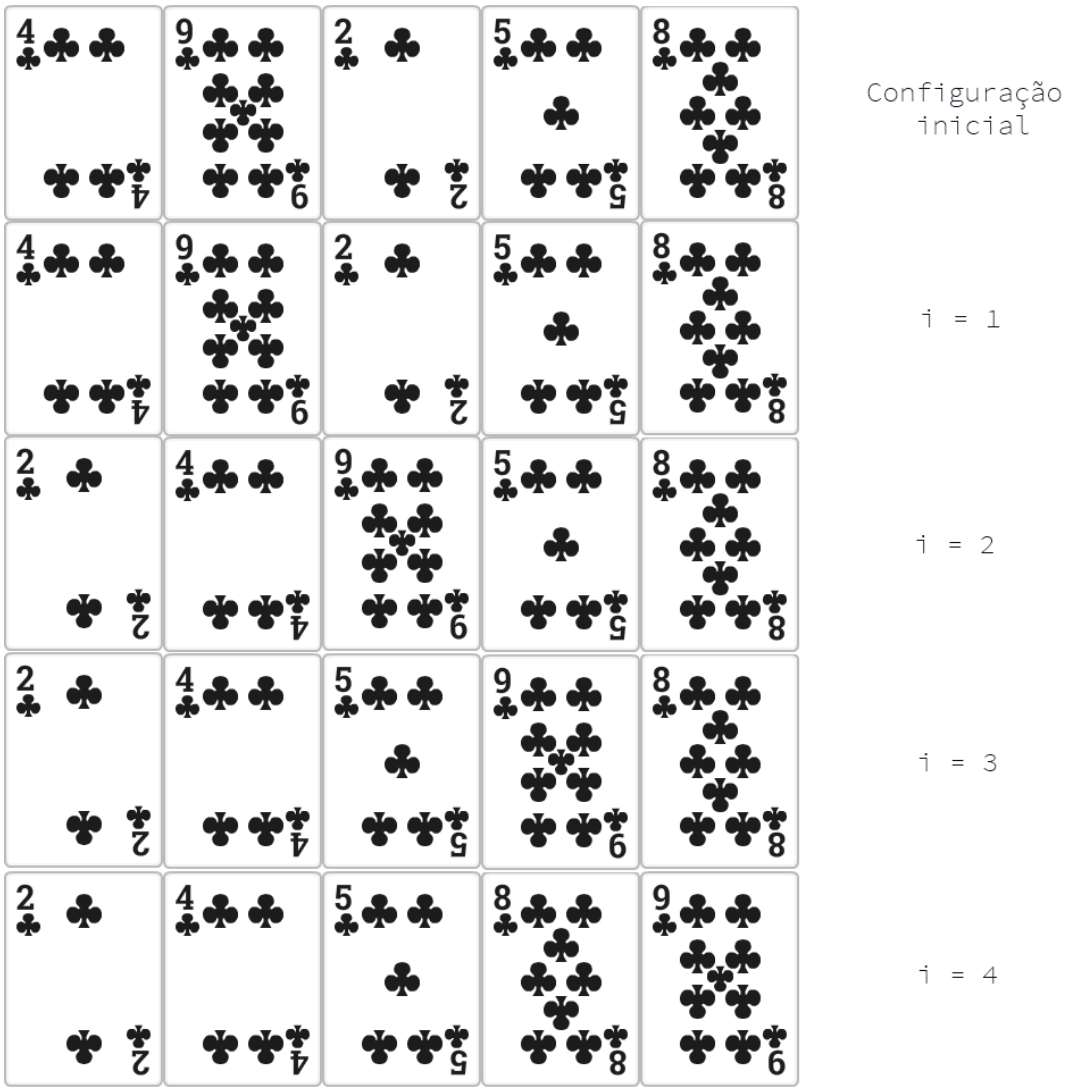

Figura 3: Ordenação de 5 cartas de baralho pelo Insertion Sort, mostrando, de cima para baixo, a disposição dos dados a cada interação

Algorithm 1 Insertion Sort(v[0..n-1],n)

```
1: for i ← 1, n − 1 do
2:     x ← v[i]
3:     j ← i − 1
4:     while j ≥ 0 and v[j] > x do
5:         v[j + 1] ← v[j]
6:         j ← j − 1
7:     end while
8:     v[j + 1] = x
9: end for
```

2.1.1 Análise de Complexidade

1. Melhor caso:

   Acontence quando o vetor $v[0..n-1]$ já está ordenado, ou seja, quando a disposição dos elementos está de forma crescente. Nesse caso, os comandos que pertencem ao segundo laço jamais serão executados, pois sempre ocorrerá $v[j] \leq x$, para qualquer j, isto é, a segunda comparação sempre será falsa. Portanto, como se acessa elemento por elemento por meio do primeiro laço (da linha 1 até 9), a complexidade do algoritmo será de $O(n)$.

2. Pior caso:

   Acontece quando a disposição dos elementos no vetor está na forma decrescente, ou seja, $v[0] \geq v[1] \geq ... \geq v[n-2] \geq v[n-1]$. Nesse caso, significa que o segundo laço passará por cada elemento de $v[0..j]$, isto é, percorrerá, necessariamente, o vetor $v[0..i-1]$ a cada iteração. Claramente, $v[0..i-1]$ é um subvetor de $v[0..n-1]$ e, assim, possui o tamanho em função de n. Portanto, o segundo laço será proporcional a $O(n)$. Finalmente, como o primeiro laço possui tempo de complexidade de $O(n)$, então, no pior caso, a complexidade será $O(n) * O(n) = O(n^2)$.

3. Caso médio:

   Conforme explorado no pior caso, o segundo laço é a operação elementar desse algoritmo e o número de iterações será em função do tamanho de n. Logo, a complexidade será também de $O(n) * O(n) = O(n^2)$.

2.2 Selection Sort

O *selection sort*, também chamado por ordenação por seleção, é um algoritmo onde dado um vetor $v[0..n-1]$, e sendo o acesso dos elementos da esquerda para a direita, a cada iteração i, usa-se uma variável auxiliar min para salvar o índice do menor elemento - inicializada pelo próprio i - e busca-se o um menor elemento no subvetor desordenado $v[i+1..n-1]$, e caso positivo, atualiza o valor de min. Finalmente, realiza-se a troca entre $v[i]$ e $v[min]$ e, no final do laço, $v[0..i]$ está ordenado. A figura 4 mostra um exemplo de ordenação a cada iteração e o pseudocódigo 2 abaixo representa a descrição desse algoritmo:

Algorithm 2 Selection Sort(v[0..n-1],n)

```
1: for i ← 0, n − 1 do
2:     min ← i
3:     for j ← i + 1, n − 1 do
4:         if v[j] < v[min] then
5:             min ← j
6:         end if
7:     end for
8:     x ← v[i]
9:     v[i] ← v[min]
10:    v[min] ← v[x]
11: end for
```

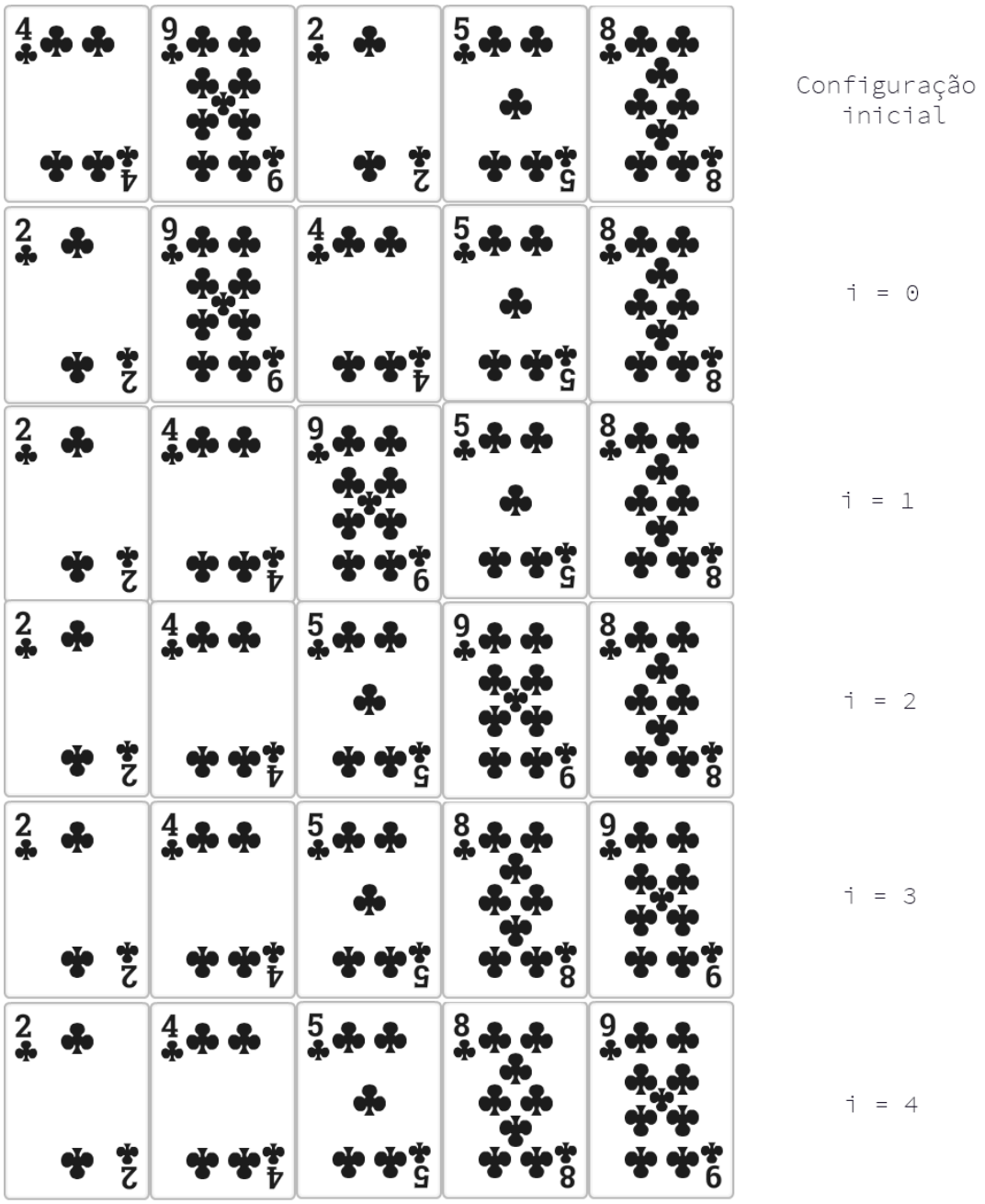

Figura 4: Ordenação de 5 cartas de baralho pelo Selection Sort, mostrando, de cima para baixo, a disposição dos dados a cada iteração

2.2.1 Análise de Complexidade

1. Melhor, pior e caso médio:

   Diferentemente da ordenação por inserção, tanto o primeiro laço(da linha 1 até 11), que percorre o vetor inteiro, como o segundo laço(da linha 3 até 7), que percorre o subvetor não ordenado, são dependentes apenas do tamanho n. Assim, irão percorrer a mesma quantidade de elementos e, portanto, inde-

pendem da disposição dos dados. Como a operação elementar se encontra na linha 4, que está dentro de um laço aninhado, a complexidade desse algoritmo será de $O(n^2)$

2.3 Bubble Sort

O Bubble Sort, conhecido também por ordenação por flutuação, é um algoritmo onde dado um vetor $v[0..n-1]$, e sendo o acesso dos elementos da esquerda para a direita, onde, a cada iteração, "flutua", ao fazer trocas com pares de elementos, o maior elemento do vetor $v[0..i]$ não ordenado para $v[i]$. Assim, no final da iteração para i, $v[i..n-1]$ está ordenado. A figura 5 mostra a ordenação para cada iteração e o pseudocódigo 3 abaixo representa a descrição desse algoritmo:

Algorithm 3 Bubble Sort(v[0..n-1],n)

```
1: for i ← n − 1, 0 do
2:     for j ← 0, i − 1 do
3:         if v[j] > v[j + 1] then
4:             x ← v[j]
5:             v[j] ← v[j + 1]
6:             v[j + 1] ← x
7:         end if
8:     end for
9: end for
```

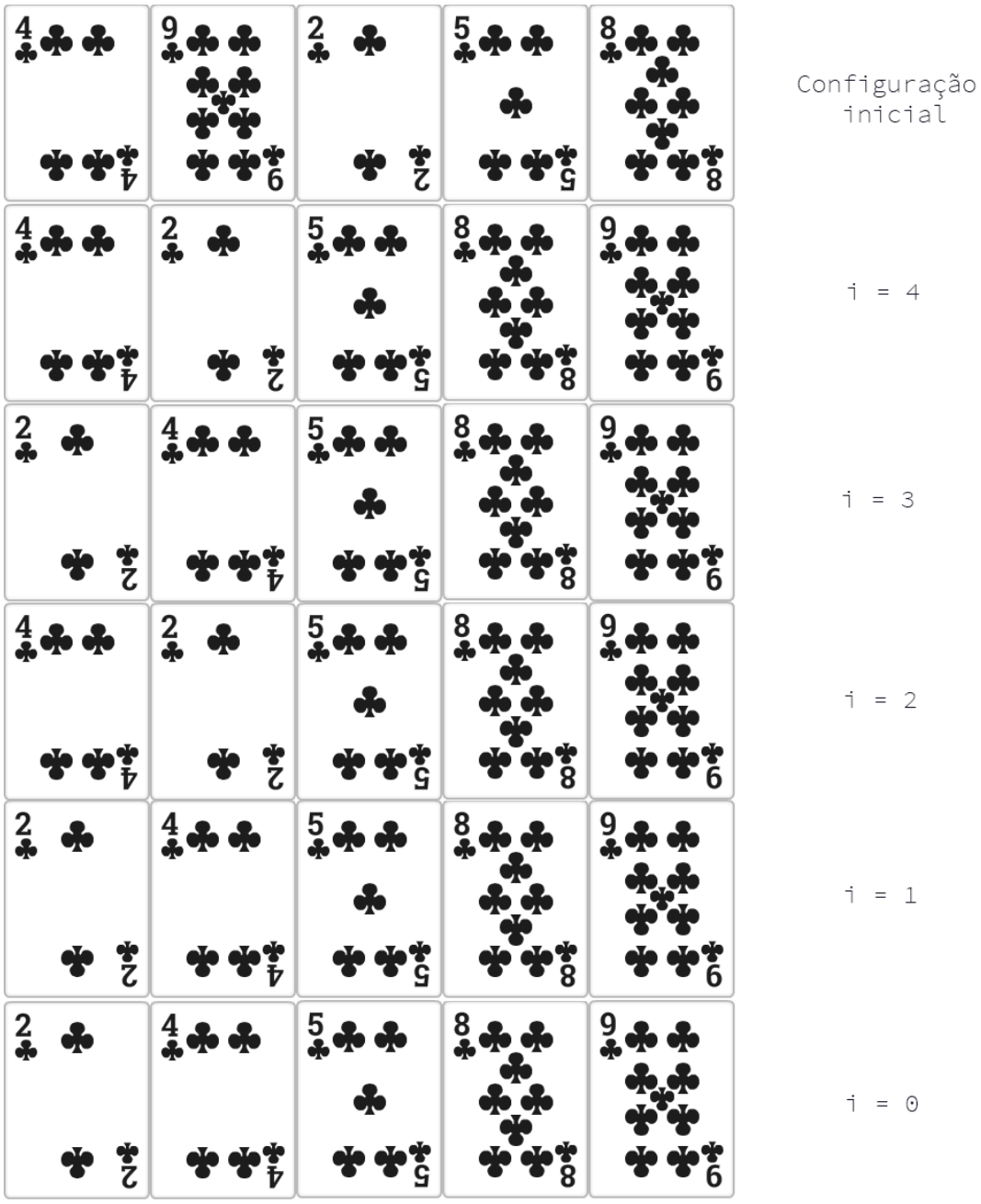

Figura 5: Ordenação de 5 cartas de baralho pelo Bubble Sort, mostrando, de cima para baixo, a disposição dos dados a cada iteração

2.3.1 Análise de Complexidade

1. Pior e caso médio: Analogamente a ordenação por seleção, os laços são dependentes apenas do tamanho n. Assim, irão percorrer a mesma quantidade de elementos e, portanto, independem da disposição dos dados. Como a operação elementar se encontra na linha 3, que está dentro de um laço aninhado, a complexidade desse algoritmo será de $O(n^2)$

2. Melhor:

   Conforme explicado para o pior e o caso médio, será também de $O(n^2)$ **caso não se use uma flag para dizer que não ocorreram mais trocas em dada iteração. Cabe observar que este trabalho não faz uso da técnica mencionada anteriormente.**

2.4 Merge Sort

O *Merge Sort* é um algoritmo que realiza a ordenação de um vetor por meio da estratégia de divisão e conquista. Essa estratégia consiste em:

1. **Divisão do problema em instâncias menores**:

   Denomina-se as variáveis auxiliares l e r onde são, respectivamente, o índice do elemento mais a esquerda e o índice do elemento mais a direita do vetor. Depois, com o uso da variável auxiliar q, calcula-se o índice que, aproximadamente, irá dividir em dois subvetores de mesmo tamanho. Essas etapas são repetidas, recursivamente, para cada subvetor, até o subvetor obtido poder ser resolvido diretamente.

2. **Resolver a instância atômica diretamente**:

   Eventualmente, o subvetor se encontrá em uma instância simples ou atômica que poderá ser ordenado diretamente. Essa situação também é usada, no algoritmo, como a base da recursão, quando a situação $l \geq r$.

3. **Combinar os resultados das diferentes instâncias**:

   Finalmente, é necessário reconstruir o vetor combinando o resultado de 2 vetores para um vetor apenas, sem perder o a propriedade de ordenação. Essa tarefa é feita pela função MERGE que, dado 2 sub vetores (em função de l, r e q), rearranja os elementos em um vetor resultante mantendo a propriedade de ordenação.

O pseudocódigo 4 representa a descrição dessa estratégia para esse algorítmo, e o pseudocódigo 5 entra em detalhes do último passo (combinação de resultados) dessa estratégia. Finalmente, a figura 6 mostra a aplicação do algoritmo a um exemplo.

Algorithm 4 MergeSort(v[0..n-1],l,r)

1: **if** $l < r$ **then**
2: $q = \lfloor (l + r)/2 \rfloor$
3: MergeSort(v,l,q)
4: MergeSort(v,q+1,r)
5: Merge(v,l,q,r)
6: **end if**

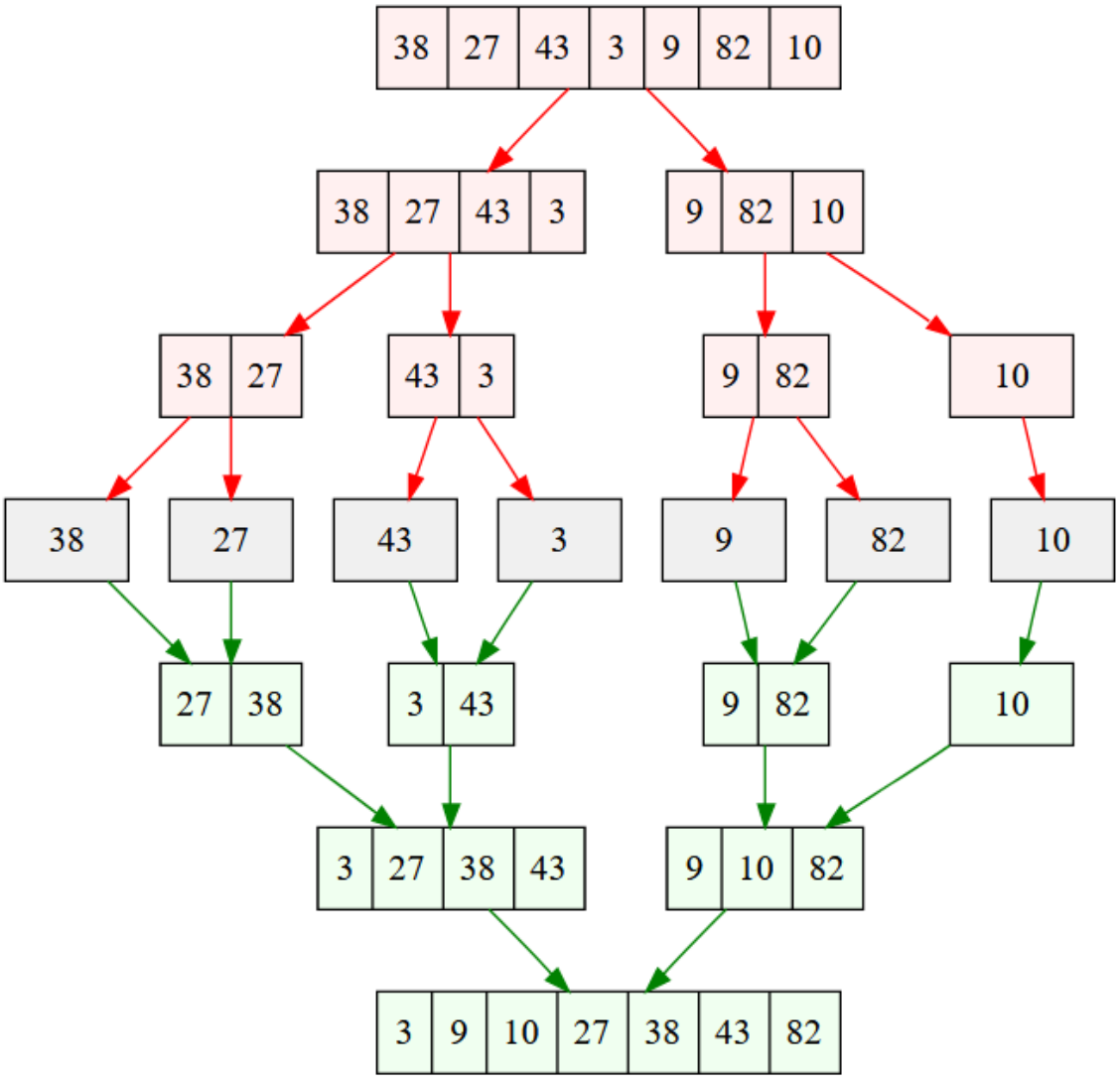

Figura 6: Ordenação de um vetor pelo Mergesort. As setas vermelhas representam a divisão em subproblemas que é realizada pelas chamadas recursivas. Uma vez que o subproblema possa ser resolvido diretamente (elementos em cinza), os elementos são recombinados e os subvetores reconstruídos pelo procedimento *Merge*, como indicam as setas verdes, até obtermos o vetor ordenado.

Algorithm 5 Merge(v[0..n-1],l,q,r)

1: $n1 \leftarrow middle - start + 1$
2: $n2 \leftarrow end - middle$
3: $v\_left \leftarrow$ cria array de tamanho $middle - start + 1$
4: $v\_right \leftarrow$ cria array de tamanho $end - middle$
5: **for** $i \leftarrow 0, n1 - 1$ **do**
6: $\quad v\_left[i] \leftarrow v[l + i]$
7: **end for**
8: **for** $j \leftarrow 0, n2 - 1$ **do**
9: $\quad v\_right[j] \leftarrow v[middle + (j + 1)]$
10: **end for**
11: $i \leftarrow 0$
12: $j \leftarrow 0$
13: $k \leftarrow l$
14: **while** $i < n1$ and $j < n2$ **do**
15: $\quad$ **if** $v\_left[i] \leq v\_right[j]$ **then**
16: $\quad\quad v[k] \leftarrow v\_left[i]$
17: $\quad\quad i \leftarrow i + 1$
18: $\quad$ **else**
19: $\quad\quad v[k] \leftarrow v\_right[j]$
20: $\quad\quad j \leftarrow j + 1$
21: $\quad$ **end if**
22: $\quad k \leftarrow k + 1$
23: **end while**
24: **while** $i < n1$ **do**
25: $\quad v[k] \leftarrow v\_left[i]$
26: $\quad i \leftarrow i + 1$
27: $\quad k \leftarrow k + 1$
28: **end while**
29: **while** $j < n2$ **do**
30: $\quad v[k] \leftarrow v\_right[j]$
31: $\quad j \leftarrow j + 1$
32: $\quad k \leftarrow k + 1$
33: **end while**
34: $free(v\_left)$
35: $free(v\_right)$

2.4.1 Análise de Complexidade

1. Melhor, pior e caso médio:

   A função MERGESORT divide, a cada nova chamada recursiva, o problema pela metade (aproximadamente), criando 2 subproblemas novos. Assim, a árvore de recorrência pode crescer até no máximo $\log_2 n$ criando n subproblemas, resultando na complexidade $O(n \log_2 n)$. Por outro lado, a função MERGE realiza a ordenação de $v[l..r]$ em complexidade $O(n)$, pois o tamanho de w é $r - l \leq n$ e os laços se encarragam de acessar apenas uma vez cada elemento desse vetor. Assim, a complexidade de MERGESORT será de $O(n \log_2 n)$ por conta do termo dominante.

O algoritmo possui perfomance melhor do que os algorítmos anteriores já estudados, porém, vale observar que o mesmo faz uso de espaço adicional de tamanho no máxino n (vetor auxiliar) para realizar a ordenação.

2.5 Quick Sort

Assim como o algoritmo anterior, o *Quick Sort* é um algoritmo que faz a ordenação de um vetor por meio de divisão e conquista. Essa estratégia consiste em:

1. **Divisão do problema em instância menores**:

   Denomina-se as variáveis auxiliares l e r onde são, respectivamente, o índice do elemento mais a esquerda e o índice do elemento mais a direita do vetor. Depois, com o uso da variável auxiliar q, atribui-se, resultado da função PARTITION, o índice que irá dividir em dois subvetores: $q - l$ e $r - (q + 1)$. Essas etapas são repetidas, recursivamente, para cada subvetor, até o subvetor obtido poder ser resolvido diretamente.

2. **Resolver a instância atômica diretamente**:

   Eventualmente, o subvetor se encontrá em uma instância simples ou atômica que poderá ser ordenado diretamente. Essa situação também é usada, no algoritmo, como a base da recursão, representada por $l \geq r$.

3. **Combinar os resultados das diferentes instâncias**:

   Finalmente, aplica-se as etapas anteriores nos dois subvetores restantes $v[l..q]$ e $v[q + 1..r]$.

O pseudocódigo 6 representa a descrição dessa estratégia para esse algorítmo. Por outro outro lado, o pseudocódigo 7 entra em datalhes na obtenção do índice de

q e da ordenação dessa estratégia. Importante notar que, no final da execução de PARTITION, $v[0..j-1] \leq x \leq v[j+1..n-1]$.

Algorithm 6 QuickSort(v[0..n-1],l,r)

```
1: if l < r then
2:     q = PARTITION(v,l,r)
3:     QUICKSORT(v,l,q-1)
4:     QUICKSORT(v,q+1,r)
5: end if
```

Algorithm 7 Partition(v[0..n-1],l,r)

```
1: x ← v[r]
2: i ← l − 1
3: for j ← l, r − 1 do
4:     if v[j] < x then
5:         i ← i + 1
6:         SWAP(v[i],v[j])
7:     end if
8: end for
9: SWAP(v[i+1],v[r])
10: return i + 1
```

2.5.1 Análise de Complexidade

1. Melhor, caso médio:

   No melhor caso, os subvetores produzidos são quase do mesmo tamanho, não passando de $\lceil n/2 \rceil$. No caso médio, a divisão dos vetores se aproxima muito mais do melhor caso do que o pior caso. Assim, a complexidade será de $O(n \log_2 n)$.

2. Pior caso:

   O pior caso acontece quando a função PARTICIONA produz um subvetor com n-1 elementos e outro que possui 1 elemento, em entradas onde vetor está ordenado ou quase ordenado. Com esse particionamento desbalanceado ocorrendo a cada nível de recursão, a complexidade será de $O(n^2)$

O algoritmo possui a mesma performance do MERGESORT que usa a mesma estratégia de divisão e conquista, com exceção do pior caso. Porém, não faz uso de espaço adicional (vetor auxiliar em função de n) para realizar a ordenação.

2.6 Heap Sort

O *Heap Sort*, faz a ordenação de um vetor por meio de uma estrutura de dados especial chamada de *heap*. Seja n o tamanho de um vetor e $v[0..n-1]$ seus n elementos. Um *heap* (binário) segue todas as seguintes propriedades:

- Dado $v[i]$, onde $0 \leq i \leq n-1$, então vale sempre que $v[i] \geq v[2i]$ e $v[i] \geq v[2i+1]$. Em outras palavras, o valor do índice pai (i) será maior ou igual ao valor de seus filhos($2i$ e $2i+1$), quando existirem.
- As folhas são inseridas "encostadas"na esquerda.
- A numeração dos elementos ocorre de cima para baixo, e da esquerda para a direita.

Naturalmente, por conta da estrutura formada representar uma árvore binária, define-se a altura de um heap como o número de arestas entre a raíz e a folha mais distante. A necessidade dessa definição se deve ao fato de que, mais adiante, será usada na análise de complexidade do algoritmo.

Entretanto, antes de construir o algorítmo HEAPSORT, será necessário definir e acrescentar mais duas funções auxiliares: MAX-HEAPIFY, BUILD-MAX-HEAPIFY. Primeiramente, definimos alguns procedimentos ou macros para o auxílio de algumas tarefas feitas pelas funções auxiliares:

1. Filho esquerdo de i:

   LEFT(i): **return** $(2i)+1$

2. Filho direito de i:

   RIGHT(i): **return** $(2i+1)+1$

3. Tamanho do Heap de v:

   HEAPSIZE(A): **return** $gHeapSize$

O último procedimento é usado para saber até qual índice do vetor estrutrutura de *heap* é seguida e, portanto, $0 \leq$ HEAPSIZE(A) $< n$. Dadas as observações descritas anteriormente, é possível construir a primeira função auxiliar chamada de MAX-HEAPIFY, como mostra o procedimento 8. Essa função verifica se o nó i está na posição correta do *heap* e, caso contrário, restaura a propriedade de heap na posição i e depois chama, recursivamente, para o índice filho trocado (*largest*). A complexidade de MAX-HEAPIFY é $O(\log_2 n)$ pois, para o pior caso, quando o *heap*

é arvore completa, seria percorrer o caminho da raiz até a folha, ou seja, a altura do *heap* ($\lfloor \log_2 n \rfloor$).

Algorithm 8 Max-Heapify(v[0..n-1],i)

```
1: l ← LEFT(i)
2: r ← RIGHT(i)
3: if l <= HEAPSIZE(v) and v[i] < v[l] then
4:     largest ← l
5: else
6:     largest ← i
7: end if
8: if r <= HEAPSIZE(v) and v[largest] < v[r] then
9:     largest ← r
10: end if
11: if i! = largest then
12:     TROCA(v[i],v[largest])
13:     MAX-HEAPFIY(v,largest)
14: end if
```

A função MAX-HEAPIFY, entretanto, é insuficiente para garantir que o vetor esteja ordenado na forma de um heap porque a chamada é feita para um índice local, não garantindo a propriedade de *heap* para outros elementos do vetor. Assim, como o próprio nome sugere, BUILD-MAX-HEAP se encarregará de construir vetor com a propriedade de *heap*, como mostra o procedimento 9, ao aplicar MAX-HEAPIFY para cada um do nós internos.

A complexidade de BUILD-MAX-HEAP é de $O(n \log_2 n)$ pois temos um laço que itera sobre uma parte de do vetor, ou seja, é em função do tamanho do vetor n e realizamos, a iteração, a operação de MAX-HEAPIFY que custa $O(\log_2 n)$.

Algorithm 9 Build-Max-Heap(v[0..n-1])

```
1: HEAPSIZE(v)← n − 1
2: for i ← ⌊(n/2)⌋ − 1, 1 do
3:     MAX-HEAPFIY(v,i)
4: end for
```

Finalmente, com as duas funções auxiliares construídas, é possível realizar a ordenação do vetor e construir o algoritmo HEAPSORT como mostra o pseucódigo 10, fazendo as seguintes operações: construímos um heap do vetor v e, em seguida, começando pelo último elemento do vetor, subistitui-se o elemento selecionado pela -

maior elemento - eliminando esse elemento do *heap* e, consequentemente, diminuindo o tamanho do *heap*. Entretanto, a substituição pode compremeter a propriedade do *heap*, então é necessário fazer a verificação por meio de MAX-HEAPFIY. Assim, dado um vetor v na iteração i, no início do laço, temos que $v[0..i]$ é um *heap* e $v[i+1..n-1]$ está ordenado.

Algorithm 10 Heapsort(v[0..n-1])

```
1: BUILD-MAX-HEAP(v)
2: for i ← n − 1, 1 do
3:     SWAP(v[0],v[i])
4:     HEAPSIZE(v)← HEAPSIZE(v) − 1
5:     MAX-HEAPFIY(v,0)
6: end for
```

2.6.1 Análise de Complexidade

1. Melhor, pior e caso médio:

   A função BUILD-MAX-HEAP, como foi visto anteriormente, é de complexidade de $O(n \log_2 n)$, então resta apenas a análise do laço que vem logo em seguida. Claramente, percorre-se o vetor inteiro menos o primeiro elemento, então temos a complexidade de $O(n)$. Nas operações dentro do laço, MAX-HEAPIFY é a operação mais custosa e, portanto, dominante. Então, $O(n) * O(\log_2 n$, resultando em $O(n \log_2 n)$. Portanto, a complexidade do algoritmo, somada as duas complexidades, será de $O(n \log_2 n)$.

   Observa-se que a disposição de dados não influencia assintoticamente no tempo de execução por conta do laço e a sua operação mais custosa, a chamada MAX-HEAPFIY .

3 Quadro de Comparação

A tabela 1 abaixo faz um resumo das complexidades dos algoritmos anteriormente estudados na seção 2, considerando a disposição do vetor de entrada, em função do seu tamanho n, destacando, em vermelho, **os casos onde a disposição dos dados afeta a complexidade assintótica do algoritmo**. É importante fazer essa distinção pois, nos testes empíricos, o foco deste trabalho está na verficação do comportamento assintótico dos algoritmos e despreza-se as pequenas variações causadas por outros termos da equação de complexidade ou devido a disposição de dados para cada um dos algoritmos.

Tabela 1 Complexidade do algoritmo em função do tamanho de entrada n e da disposição de dados.

Algoritmo \ Caso	Melhor	Pior	Médio
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick Sort	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$
Heap Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

A tabela 2 a seguir mostrar qual a disposição de dados para se obter o melhor e o pior caso para cada algoritmo:

Tabela 2 Disposição de dados para cada algoritmo no pior (inverso) ou melhor caso (ordenado).

Algoritmo \ Caso	Melhor	Pior
Insertion Sort	crescente	decrescente
Selection Sort	crescente	decrescente
Bubble Sort	crescente	decrescente
Merge Sort	crescente	decrescente
Quick Sort	entrada tal que divide em 2 sub-vetores de tamanhos iguais	crescente ou quase crescente
Heap Sort	decrescente	crescente

4 Ambiente de execução

As configurações do computador utilizado para fazer a execução do experimento foram as seguintes:

- **Processador:**

  2,7 GHz Intel Core i5

- **Memória:**

  8 GB 1867 Mhz DDR3

- **Sistema Operacional:**

  macOS Mojave Versão 10.14.1

5 Resultados e análise

5.1 Resultados

As tabelas a seguir mostram o tempo gasto para a ordenação de um vetor em função do tamanho de entrada para cada algoritmo. Para cada tamanho de entrada, foi cronometrado o tempo, em segundos, para três disposições de dados: aleatório, ordenado (melhor caso) e inverso (pior caso). Cabe destacar que a disposição de dados diferem de algoritmo para algoritmo, para cada uma das três situações, conforme discutido na seção 2 e resumido na seção 3. Uma observação pertinente, na tabela 7, a falta de tempos registrados no último tamanho de entrada se deve ao acontecimento de estouro da pilha.

Tabela 3 Comparação de tempo do Insertion Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	0.360947	0.000289	0.705970
40000	1.393672	0.000578	2.786891
80000	5.533182	0.001208	11.127188
160000	22.187538	0.002303	44.368162
320000	90.314721	0.004669	187.621580

Tabela 4 Comparação de tempo do Bubble Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	1.964841	0.573983	2.004713
40000	5.982058	2.271881	10.906751
80000	31.813755	9.089173	43.932555
160000	133.071623	36.113112	172.643548
320000	531.812667	144.499940	700.991444

Tabela 5 Comparação de tempo do Selection Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	0.501030	0.497501	0.529155
40000	1.982005	1.977815	2.125196
80000	7.847505	7.865431	8.358270
160000	31.166242	31.186192	33.301231
320000	124.440248	124.656954	133.303778

Tabela 6 Comparação de tempo do Merge Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	0.006117	0.005269	0.006663
40000	0.011804	0.010714	0.011376
80000	0.020005	0.015922	0.018636
160000	0.035854	0.024309	0.027441
320000	0.065972	0.047276	0.046186

Tabela 7 Comparação de tempo do Quick Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	0.009229	2.572089	1.522332
40000	0.013365	10.226271	6.269351
80000	0.022033	41.956168	25.022830
160000	0.042468	167.315238	100.298074
320000	0.086821	-	-

Tabela 8 Comparação de tempo do Heap Sort em segundos

N	Aleatório	Ordenado	Inverso
20000	0.013097	0.011009	0.009731
40000	0.016329	0.017992	0.018287
80000	0.029267	0.030127	0.027826
160000	0.063039	0.053699	0.052497
320000	0.131687	0.103375	0.102514

5.2 Análise

Com base nos dados obtidos dos testes empíricos realizados na seção anterior, foi criada uma série de figuras que representam o comportamento desses algoritmos. A figura 7 mostra o gráfico de comparação entre os algoritmos descritos na seção 2 para entrada aleatória de dados. Uma questão que fica evidente é a relevância da notação assintótica para descrever a eficiência do algortimo quando trata-se de valores de entrada grandes, contudo quando observado para entradas pequenas (menos que 50000) os algortimos tem similaridade no desempenho. Além disso, fica muito claro o péssimo desempenho dos algoritmos de ordenação elementares (Insertion Sort, Bubble Sort e Selection Sort) a medida que o tamanho da entrada aumenta.

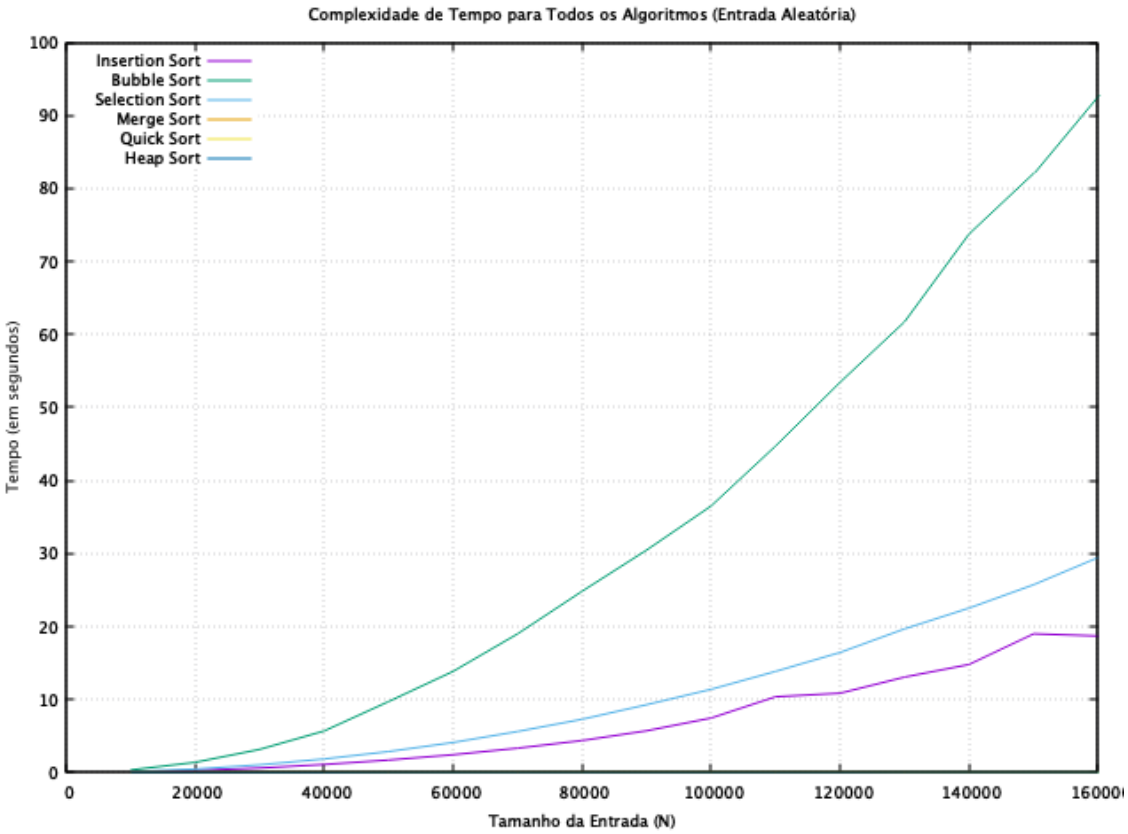

Figura 7: Gráfico da complexidade dos algoritmos estudados com entrada aleatória.

A figura 8 mostra o gráfico de comparação entre os algoritmos descritos na seção para a entrada ordenada. Observar-se uma grande ganho de eficiência para o Insertion Sort mas um gasto maior de tempo do Quick Sort, como era de se esperar.

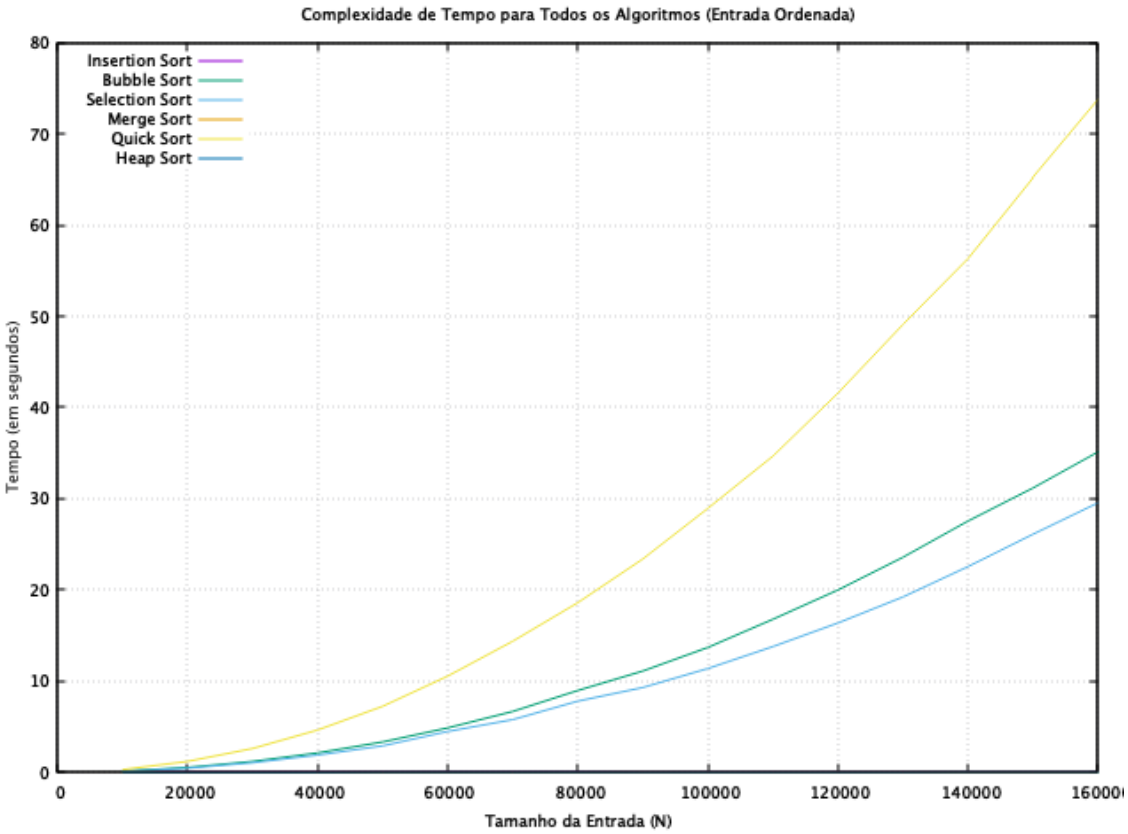

Figura 8: Gráfico da complexidade dos algoritmos estudados com entrada ordenada.

A figura 9 mostra o gráfico de comparação entre os algoritmos descritos na seção para a entrada inversa. Observa-se que, apesar de um tempo maior se comparado aos resultados das entradas aleatórias, o comportamento assintótico se mantém.

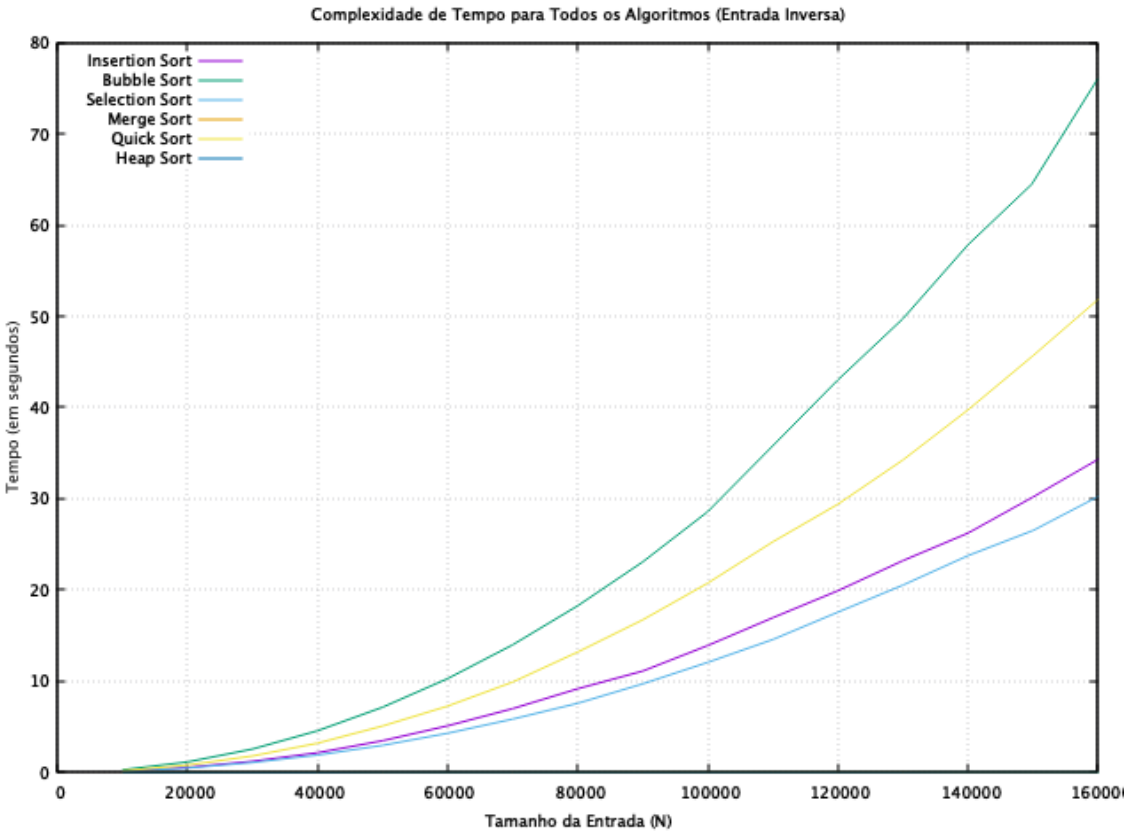

Figura 9: Gráfico da complexidade dos algoritmos estudados com entrada inversa.

Quando comparamos os algoritmos de ordem linearítmica, como mostra a figura 10, podemos notar que o Heap Sort se destaca dos demais, isso se dá devido as multiplicativas pequenas que o algortimo executa. Contudo, neste caso, podemos notar que o Merge Sort ainda assim consegue ser mais rápido devido a distribuição dos dados.

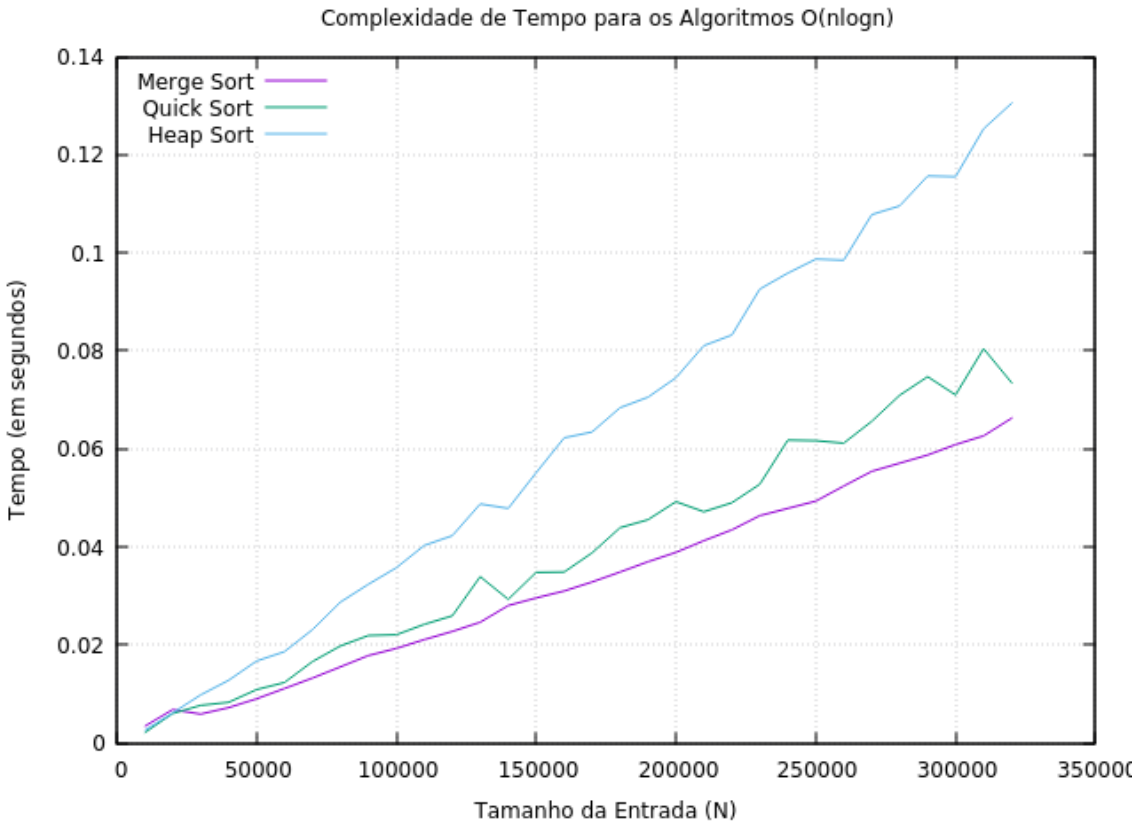

Figura 10: Gráfico da complexidade dos algoritmos linearítmicos estudados para entradas aleatória.

6 Conclusão

A ordenação de elementos de um vetor é um dos problemas mais estudados dentro da Ciência da Computação. A necessidade de estudar o tempo de execução e, principalmente, a complexidade dos algoritmos é crucial para prever seu funcionamento e avaliar seu uso na prática.

Nesse trabalho, foi estudado cada algoritmo selecionado - algoritmos elementares, Mergesort, Quicksort e Heapsort -, explorando seu funcionamento e, consequentemente, a complexidade do tempo, levando em consideração a influência do tamanho do vetor e sua disposição. As complexidades assintóticas obtidas foram resumidas em um quadro e foram validadas, empiricamente, após análise de execução no ambiente de proposto.

7 Referências

1. Bibliografia: T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third edition, MIT Press, 2009.

2. Imagens:

   - Exemplo do Merge Sort: Wikipedia
   - Cartas (individualmente): Boardgame Pack

3. Repositório: https://github.com/dmb42odyssey/AA2019