[2]Departamento de Engenharia da Computação e Sistemas

Universidade Estadual do Maranhão

Davi M. Carvalho[1]

[2]

1. Algoritmos Escolhidos

O intuito do trabalho foi a realização da averiguação da complexidade de algoritmos de busca, logo fez-se necessária a seleção de três algoritmos, que foram escolhidos motivados principalmente por serem utilizados como base e amplamente conhecidos. Esses são descritos juntamente com suas complexidades (melhor, médio e pior casos), uso de memória e estabilidade, através da Tabela 1.

Algoritmo	Melhor	Médio	Pior	Memória	Estável
In-Place Merge Sort	n $\log_2 n$	n $\log_2 n$	n $\log_2 n$	1	Sim
Shelsort	n $\log_2 n$	n $\log_2 n$	n $\log_2 n$	n $\log_2 n$	Não
Libraysort	n $\log n$	n $\log n$	n^2	n	Não

Quanto a autoria dos códigos, já que houve a possibilidade de copiar, foi utilizado o chatGPT que disponibilizou os mesmos.

1.1. Explicação Estrutura dos Códigos

```
1  public interface Exemplo {
2      public void order(int[] array) //execacao do algoritmo
           de ordena  o
3      int[] averageCase(int n) //retorna caso medio, como ele
           semelhante aos outros, ele    uma implementa o
       a parte
4      int[] worstCase(int n) //Retorna pior caso
5      int[] bestCase(int n) // retorna melhor caso
6  }
```

```
1   protected int[][] avarageCase(Integer qtt) {
2       int[][] nums = new int[5][qtt];
3       Random r = new Random();
4       for (int i = 0; i < 5; i++) {
5           for (int j = 0; j < qtt; j++) {
6               nums[i][j] = r.nextInt(1_000_001);
7           }
8       }
9       return nums;
10  }
11
12  protected void setMemoryUsed() {
13      Long currentMemoryUsed = ((Runtime.getRuntime().
           totalMemory()) - Runtime.getRuntime().freeMemory());
14      memoryUsed  = currentMemoryUsed > memoryUsed?
           currentMemoryUsed : memoryUsed;
```

```
    }

    @Override
    public long getHigherMemory() {
        return memoryUsed;
    }
```

Além do setMemoryUsed que será utilizado entre as linhas do código para tentar capturar o ponto de maior pico de mmeória e getHighertMemory que é apenas um get para pegar a maior memória encontrada. Na subseção 1.2 será descrita o primeiro algoritmo.

1.2. Shellsort

Apresentação do código:

```
    public class Shellsort extends AlgorithmImpl {

    public int[] order(int[] nums) {
        int h = 1;
        int n = nums.length;
        while (h < n) {
            h = h * 3 + 1;
        }
        h = h / 3;
        int c, j;
        while (h > 0) {
            for (int i = h; i < n; i++) {
                c = nums[i];
                j = i;
                while (j >= h && nums[j - h] > c) {
                    nums[j] = nums[j - h];
                    j = j - h;
                }
                nums[j] = c;
            }
            setMemoryUsed();
            h = h / 2;
        }
        return nums;
    }


    @Override
    public int[][] worstCase(Integer qtt) {
        int[][] nums = new int[1][qtt];
        for (int i=qtt; i > 0; i--)
            nums[0][qtt-i] = i;
        return nums;
    }

```

```
36      @Override
37      protected int[][] bestCase(Integer qtt) {
38          int[][] nums = new int[1][qtt];
39          for (int i = 0; i < qtt; i++) {
40              nums[0][i] = i;
41          }
42          return nums;
43      }
```

Cálculo de complexidade através de expansão.

A sequência de Knuth é: $h = 1, 4, 13, \ldots$, e para cada h realizamos insertion sort com custo: $\mathrm{O}\left(\frac{n^2}{h}\right)$

Somando para todos os gaps:

T(n) = $\sum_h O\left(\frac{n^2}{h}\right)$ Somando para todos os gaps:

T(n) = $\sum_h O\left(\frac{n^2}{h}\right)$

Como o número de gaps é $O(\log n)$ e a soma converge:

T(n) = O($n^{3/2}$)

1.3. Librarysort

Apresentação do código:

```
1       public int[] order(int[] nums) {
2           if (nums.length <= 1) return nums;
3
4           double epsilon = 0.5; // define quanto espa o extra
                reservado
5           int n = nums.length;
6           int size = (int) ((1 + epsilon) * n);
7           Integer[] gaps = new Integer[size];
8           Arrays.fill(gaps, null);
9
10          // Insere o primeiro elemento no meio
11          int mid = size / 2;
12          gaps[mid] = nums[0];
13
14          int count = 1; // n mero de elementos inseridos
15
16          for (int i = 1; i < n; i++) {
17              int val = nums[i];
18
19              // Busca bin ria para posi  o
20              int pos = binarySearch(gaps, val);
21
22              // Move para abrir espa o se necess rio
23              pos = findGap(gaps, pos);
24
```

```
25              // Insere
26              gaps[pos] = val;
27              setMemoryUsed();
28              count++;
29
30              // Se muitos elementos inseridos, reespalha
31              if (count > gaps.length * (1.0 / (1.0 + epsilon))) {
32                  gaps = rebalance(gaps, epsilon);
33                  setMemoryUsed();
34              }
35          }
36
37          // Copia os elementos de volta
38          int index = 0;
39          for (Integer val : gaps) {
40              if (val != null) {
41                  nums[index++] = val;
42              }
43          }
44          return nums;
45      }
46      private static int binarySearch(Integer[] gaps, int val) {
47          int low = 0, high = gaps.length - 1;
48          while (low <= high) {
49              int mid = (low + high) / 2;
50              if (gaps[mid] == null || gaps[mid] > val) {
51                  high = mid - 1;
52              } else if (gaps[mid] < val) {
53                  low = mid + 1;
54              } else {
55                  return mid;
56              }
57          }
58          return low;
59      }
60
61      private static int findGap(Integer[] gaps, int pos) {
62          // Busca posi  o dispon vel    direita
63          while (pos < gaps.length && gaps[pos] != null) {
64              pos++;
65          }
66          if (pos >= gaps.length) {
67              // Se n o encontrou    direita, procura    esquerda
68              pos = pos - 1;
69              while (pos >= 0 && gaps[pos] != null) {
70                  pos--;
71              }
72          }
73          return pos;
```

```
74      }
75
76      private static Integer[] rebalance(Integer[] gaps, double
            epsilon) {
77          int n = (int) Arrays.stream(gaps).filter(x -> x != null)
                .count();
78          int newSize = (int) ((1 + epsilon) * n);
79          Integer[] newGaps = new Integer[newSize];
80          Arrays.fill(newGaps, null);
81
82          int idx = 0;
83          int step = newSize / n;
84          for (Integer val : gaps) {
85              if (val != null) {
86                  newGaps[idx] = val;
87                  idx += step;
88              }
89          }
90          return newGaps;
91      }
92
93      @Override
94      public int[][] worstCase(Integer qtt) {
95          int[][] nums = new int[1][qtt];
96          for (int i = 0; i < qtt; i++) {
97              nums[0][i] = qtt - i;
98          }
99          return nums;
100     }
101
102     @Override
103     protected int[][] bestCase(Integer qtt) {
104         int[][] nums = new int[1][qtt];
105         for (int i = 0; i < qtt; i++) {
106             nums[0][i] = i;
107         }
108         return nums;
109
110     }
111 }
```

Cálculo de complexidade através de expansão. Cada inserção faz busca binária e possivelmente rebalanceamento. A recorrência é:

$$T(n) = T(n-1) + \log n$$

Expandindo: $T(n) = T(n-1) + \log n$

$$= T(n-2) + \log(n-1) + \log n$$

$$\vdots$$

$$= \sum_{i=2}^{n} \log i = \log(n!)$$

Pelo uso da fórmula de Stirling: $\log(n!) = \Theta(n \log n)$

Portanto: T(n) = O(n $\log n$)

1.4. In-Place MergeSort

```java
public class InPlaceMergeSort extends AlgorithmImpl {

    @Override
    public int[] order(int[] nums) {
        int len = nums.length;
        return mergeSort(nums, 0, len-1);
    }

    public int[] merge(int[] nums, int start, int mid, int end)
       {
        setMemoryUsed();
        int start2 = mid + 1;

        // Se os elementos j  estiverem em ordem, n o faz nada
        if (nums[mid] <= nums[start2]) {
            return nums;
        }

        while (start <= mid && start2 <= end) {
            if (nums[start] <= nums[start2]) {
                start++;
            } else {
                int value = nums[start2];
                int index = start2;

                // Desloca todos os elementos entre start e
                   start2 para a direita
                while (index != start) {
                    nums[index] = nums[index - 1];
                    index--;
                }

                nums[start] = value;

                // Atualiza os ponteiros
                start++;
                mid++;
                start2++;
            }
        }
        return nums;
    }

    public int[] mergeSort(int[] arr, int l, int r) {

```

```
44          if (l < r) {
45              int m = l + (r - l) / 2;
46
47              mergeSort(arr, l, m);
48              mergeSort(arr, m + 1, r);
49
50              this.merge(arr, l, m, r);
51              setMemoryUsed();
52          }
53          return arr;
54      }
55
56      @Override
57      public int[][] worstCase(Integer qtt) {
58          int[][] nums = new int[1][qtt];
59          for (int i = 0; i < qtt; i++) {
60              nums[0][i] = qtt - i;
61          }
62          return nums;
63      }
64
65      @Override
66      protected int[][] bestCase(Integer qtt) {
67          int[][] nums = new int[1][qtt];
68          for (int i = 0; i < qtt; i++) {
69              nums[0][i] = i;
70          }
71          return nums;
72
73      }
74  }
```

Cálculo de complexidade através de expansão. Definimos a recorrência para o número de comparações como:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Assumindo que $n = 2^k \Rightarrow k = \log_2 n$.

Expandindo: $T(2^k) = 2T(2^{k-1}) + 2^k$

$$\begin{aligned}
&= 2(2T(2^{k-2}) + 2^{k-1}) + 2^k \\
&= 4T(2^{k-2}) + 2 \cdot 2^k \\
&= 8T(2^{k-3}) + 3 \cdot 2^k \\
&\vdots \\
&= 2^k T(1) + k \cdot 2^k \\
&= k \cdot 2^k = n \log n
\end{aligned}$$

Logo, a complexidade de tempo é:

$T(n) = O(n \log n)$

Em termos de deslocamentos in-place, o pior caso pode chegar a $O(n^2)$.

2. Testes realizados

Os testes sugeridos foram os casos com [10,100,1000,10000,1000000,100000000] (entre dez a dez milhões). Os resultados foram salvos através de um arquivo csv que retornou o resultado. A tabela 1 representa os 5 primeiros resultados de cada algoritmo.

Table 1. Resultados de execução dos algoritmos de ordenação

Data	Algoritmo	N	Caso	Tempo (ms)	Memória (bytes)
2025-06-01T13:04:14	Shellsort	10	pior	3,443,784	61,699
2025-06-01T13:04:14	Shellsort	10	melhor	2,923,904	23,594
2025-06-01T13:04:14	Shellsort	10	médio	2,925,147	25,096
2025-06-01T13:04:14	Shellsort	100	pior	2,925,120	97,611
2025-06-01T13:04:15	Shellsort	100	melhor	2,925,120	95,063
2025-06-02T08:37:21	Library	10	pior	3,439,456	132,033
2025-06-02T08:37:21	Library	10	melhor	2,924,040	55,256
2025-06-02T08:37:22	Library	10	médio	2,925,283	49,412
2025-06-02T08:37:22	Library	100	pior	2,925,304	514,221
2025-06-02T08:37:22	Library	100	melhor	2,925,304	374,195
2025-06-02T08:37:23	Library	100	médio	2,927,131	201,446
2025-06-03T20:33:59	In-Merge Sort	10	pior	3,440,040	88,461
2025-06-03T20:33:59	In-Merge Sort	10	melhor	2,923,920	42,108
2025-06-03T20:34:00	In-Merge Sort	10	médio	2,925,163	43,097
2025-06-03T20:34:00	In-Merge Sort	100	pior	2,925,184	458,702
2025-06-03T20:34:00	In-Merge Sort	100	melhor	2,925,184	178,481

References

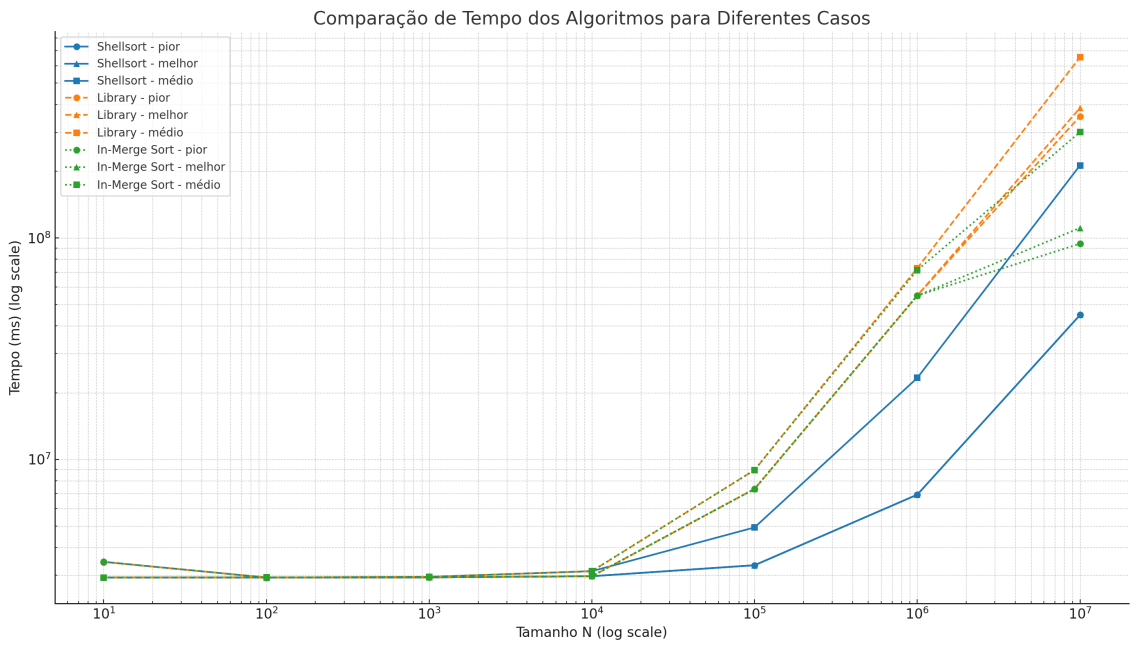

Figure 1. Gráfico resultados algoritmos por entradas.