Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Jonatan Henrique da Silva - 5205

"ANÁLISE DE ALGORITMOS DE ORDENAÇÃO"

Rio Paranaíba - MG

2022

Universidade Federal de Viçosa

Campus **Rio Paranaiba**

Jonatan Henrique da Silva - 5205

"ANÁLISE DE ALGORITMOS DE ORDENAÇÃO"

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2022

RESUMO

Os algoritmos de ordenação possuem como objetivo trazer, além de implementações que buscam serem mais eficientes na resosulçao de vetores desordenados, uma reflexão sobre quais estruturas são mais eficientes na realização desta resolução, onde custo computacional e tempo de execução são os fatores mais importantes na medição da eficiencia destes algoritmos. Com isso, alguns exemplos como Shell Sort e Merge Sort, demonstram ser algoritmos que buscam ter maior eficiencia porém, ainda sim, algoritmos como o Bubble Sort, Insertion Sort e Selection Sort, mesmo que em tese demonstram ser eficientes, possuem menor desempenho comparado a outros e algoritmos que têm objetivos de trazer maior fluidez na ordenação dos elementos, possuem fraquezas como a instabilidade de execução do algoritmo, como é o caso do Quick Sort.

Sumário

1 INTRODUÇÃO

Quando deparado com estruturas simples de fácil resolução, a primeira coisa que é refletida é o tamanho da complexidade desta, sendo revelado que não se trata de uma coisa simples apenas, mas com um histórico de descoberta e solução.

Na implementação dos softwares, entende-se que cada estrutura demanda um custo para que seja executada, seja ela um custo menor, na qual, leva-se a ter uma eficiencia maior ou que exija um custo maior para que garanta a plena execução daquilo que foi desenvolvido. A eficiencia semmpre foi um dos desafios da computação, principalmente, levando em consideração a capacidade tecnológica da época, onde não havia-se um grande poder de processamento e uma estrutura de grande capacidade de armazenamento volátil.

Os algoritmos entram nessa linha de raciocínio, surgindo implementações com o objetivo de ser mais eficiente no propósito na qual foi concebido. Surgindo daí, diversas formas de implementações, sejam elas melhoradas ou completamente reformuladas, trazendo sempre o equilíbro entre custo e eficiencia.

Com os algoritmos de ordenação, as variadas implementações com características e comportamentos particulares e nomináveis, leva a uma análise das vantagens que cada algoritmo pode oferecer, podendo servir para futuras melhorias ou inspiração para novas formas implementáveis. Assim, os algoritmos Insertion e Bubble, partem do princípio de haver verificações repetidas dos seus vetores para que sejam feitas as operações, com as diferenças de que o Bubble verifica a posição da frente e o Insertion verifica as posições anteriores ao elemento-chave. Já o Selection Sort, parte do princípio de encontrar o menor valor, para que, assim seja feitas as operações. O Shell Sort é uma ramificação do Insertion, porém ele subdivide os vetores primeiramente, para que assim, possa realizar as verificações de forma que não gere tantas repetições para que seja ordenado, fazendo com que as operações sejam quase que simultaneas.

Ainda na busca por implementações mais eficientes, a recursividade demonstrou que era possível trazer uma eficiência superior à outros tipos de implementações, na qual, se exigia muitas ações para que pudesse serem feitas as ordenações corretas dos elementos. Com isso, o Merge Sort e o Quick Sort veio com a idéia de dividir para conquistar *(Divide and Conquer)*,

para que pudesse separar o vetor em duas subpartes recursivamente e que sejam feitas suas ordenações, porém com as diferenças em que o primeiro lida com as separações e depois as ordenações combinadas e o segundo lida com a ordenação por meio do particionamento do vetor fazendo com que sejam separadas, por meior de um pivô, os números maiores e menores.

Com isso, é visualizado um cenário, na qual, os seis algoritmos citados, são analisados e comparados minuciosamente pelas suas respectivas vantagens e desvantagens.

2 ALGORITMO INSERTION SORT

O intuito do algoritmo é fazer a ordenações de arrays que se encontram se encontram desordenados, e busca uma forma implementável faça com que seja ordenada seus elementos. Na prática, o algoritmo se depara com um array com n posições, necessitando de *loops* e uma chave na qual possui o valor do elemento na posição atual do array dentro do *loop*.

Para que seja feita a ordenação, é necessário que o *loop* do array seja iniciado na segunda posição, sendo feita uma comparação repetida entre o elemento da posição atual e o elemento que está inserido na chave, para identificar o menor valor dentro do array e, assim, efetuar a ordenação. Enquanto é efetuado essa comparação dentro do *loop*, todos os outros elementos antecedentes ao elemento que se tornou chave tomam posições superiores no array. Quando o menor elemento é encontrado, ele é colocado na posição na qual o menor foi encontrado no array, como o exemplo na Figura 1.

O pseudocódigo exemplificado na Figura 2 dá uma noção de como o algoritmo age para que seja feita a ordenação, gerando uma característica específica que serve de inspiração para o nome do algoritmo.

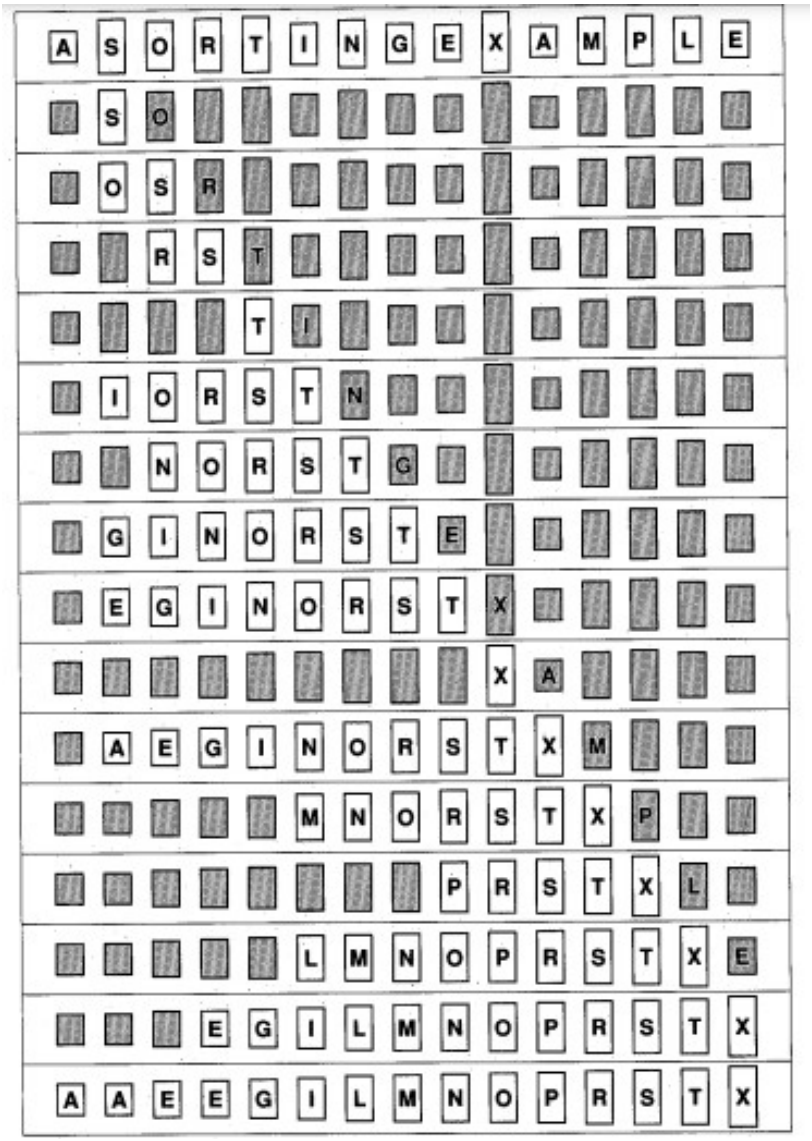

Figura 1: Exemplo de Execução do Algoritmo Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Figura 2: Exemplo de pseudocódigo do Insertion Sort

3 ALGORITMO BUBBLE SORT

Como o nome sugere, o algoritmo lida com elementos que teoricamente fazem parte de uma "bolha", pois são comparados dois elementos por vez e efetuando suas respectivas trocas.

O algoritmo lida na prática com um array de n posições que são feitos por meio de dois *loops*, na qual, um *loop* percorre todo o array de forma incremental e o outro *loop* percorre o array até n - i, onde, i é o índice que o *loop* anterior se encontra. Por exemplo: Se o array possui 10 posições e o *loop* encontra-se no índice 7, o próximo *loop* que seria a próxima instrução deste *loop* percorreria apenas 3 posições.

Então é feita uma verificação na qual o elemento-chave compara com o elemento vizinho, efetuando a troca, caso o elemento vizinho seja menor que o elemento-chave. O que pode ser visto o exemplo na figura 3.

O pseudocodigo da figura 4 dá um entendimento maior de como se comporta o algoritmo, exemplificando melhor o que se daria o conceito "bolha".

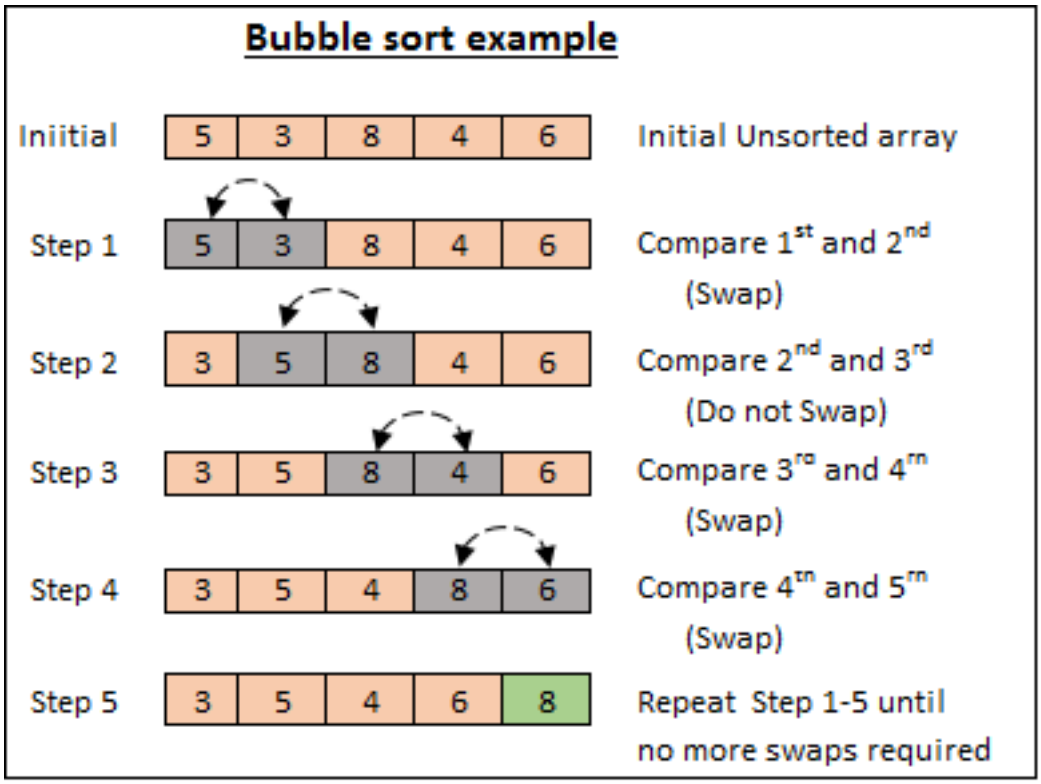

Figura 3: Exemplo de execução do algoritmo Bubble Sort

```
1. bubblesort(A[1...n], n)
2. |   para i ← 1 até n - 1
3. |   |   para j ← 1 até n - i
4. |   |   |   se(A[j] > A[j + 1])
5. |   |   |   |   trocar(A, j, j + 1)
6. |   |   |   fim_se
7. |   |   fim_para
8. |   fim_para
9. fim_bubblesort
```

Figura 4: Pseudocodigo do algoritmo Bubble Sort

4 ALGORITMO SELECTION SORT

A idéia por trás do algoritmo é procurar pelo menor valor do array, assim "selecionando"o valor para que seja este seja movimentado para a primeira posição do array, fazendo com que a primeira posição do array seja a posição subsequente.

O algoritmo inicia com uma instrução *loop* onde esta passa pelo array de posições n até $n-1$. Com isso, é adicionado o índice da posição em que o primeiro *loop* entrando-se, assim, no segundo *loop* onde este irá fazer a varredura pelo resto do vetor encontrando o índice do menor. Ao terminar a varredura, é comparado o valor que está na posição do índice do primeiro *loop* se ele é diferente do menor valor que foi encontrado pelo índice no segundo *loop*, efetuando a troca como na figura 5, se este for verdade, incrementando o valor do primeiro *loop* fazendo com que o índice que fora iniciado não seja mais acessado.

O pseudocodigo do Selection Sort na figura 6 mostra a fundo quais instruções são utilizadas para que seja efetuada a seleção do menor valor e como a troca é feita.

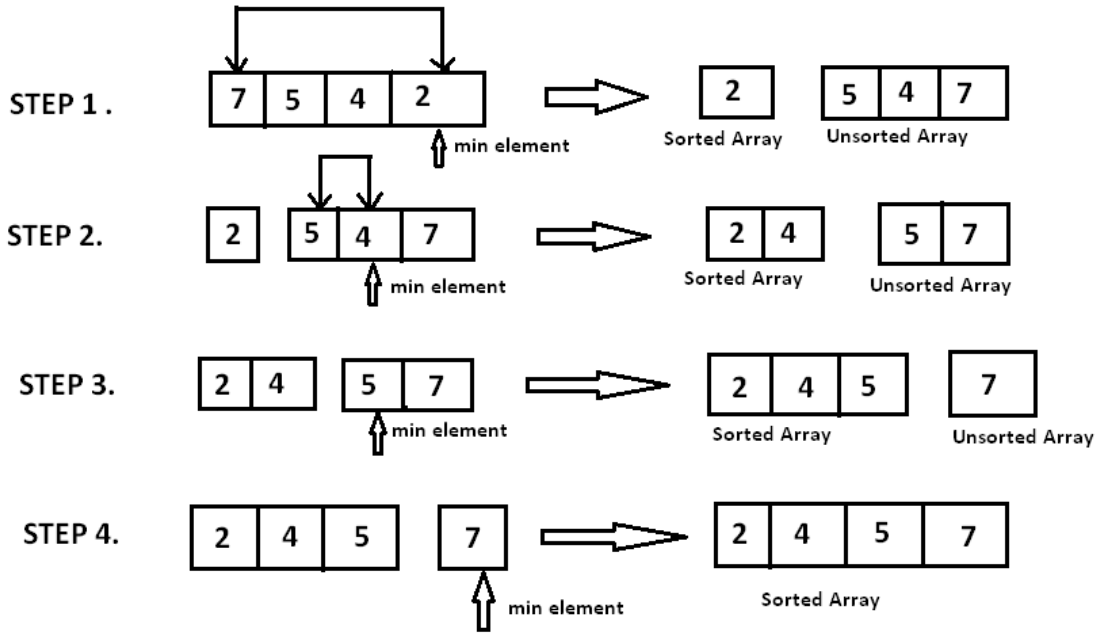

Figura 5: Exemplo de execução do Algoritmo Selection Sort

```
01. var i,j,aux,menor : INTEIRO
02. PARA i=0 ATE (vet.tamanho-1) PASSO 1
03.  menor = i
04.  PARA j=i+1 ATE vet.tamanho PASSO 1
05.   SE (vet[menor] > vet[j]) ENTAO
06.    menor = j
07.   FIMSE
08.  FIMPARA
09.  SE (menor != i) ENTAO
10.   aux = vet[menor]
11.   vet[menor] = vet[i]
12.   vet[i] = aux
13.  FIMSE
14. FIMPARA
```

Figura 6: Pseudocodigo do algoritmo Selection Sort

5 ALGORITMO SHELL SORT

Este algoritmo é primeiramente definido como uma variação do Insertion Sort, pois ele utiliza-se do conceito de movimentação dos elementos no array de forma que o elemento-chave seja movido para a posição onde há o menor valor possível, com isso, é possível perceber um problema quando o algoritmo tem que lidar com arrays maiores. Com o Shell Sort, a idéia é justamente lidar com estes elementos distantes de forma este array seja divido em subarrays na qual são feitas as trocas nesse subarray, subdividindo então este subarray, assim, sucessivamente como na figura 7.

Na prática, o algoritmo primeiramente encontra, por meio de um *loop*, o valor h na qual servirá para subdividir o array. Depois de terminado o *loop* anterior, o algoritmo entra em um outro *loop* na qual é verificado se este valor h é maior do que 0, então entra-se num segundo *loop* que partirá do valor h até n posições do array, pegando como chave o primeiro elemento do *loop* e armazenando a posição j deste elemento no array, assim, é feita a comparação se a posição j do elemento é maior que $h - 1$ e o elemento-chave é menor ou igual ao elemento na posição $j - h$ por meio de um *loop*. Assim, se for verdade, o elemento na posição $j = h$ é trocado com o elemento da posição j que nada mais é o elemento-chave como demonstrado no pseudocodigo da figura 8.

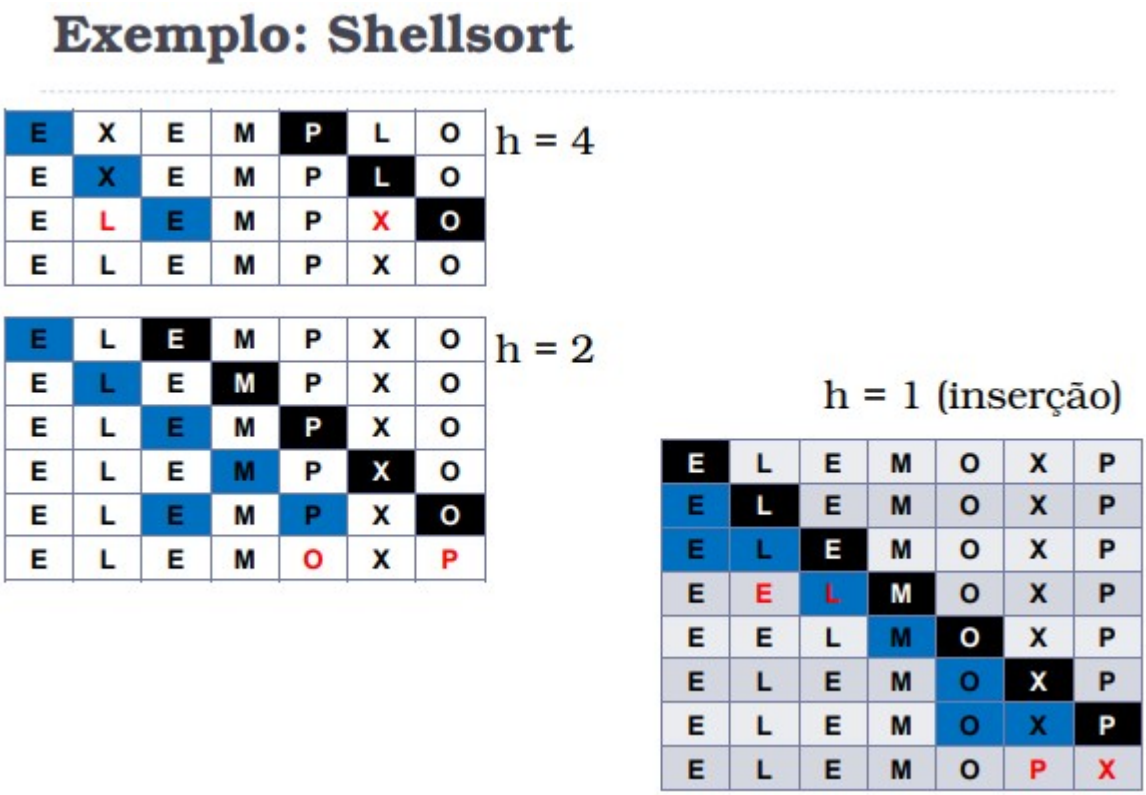

Figura 7: Exemplo de execução do algoritmo Shell Sort

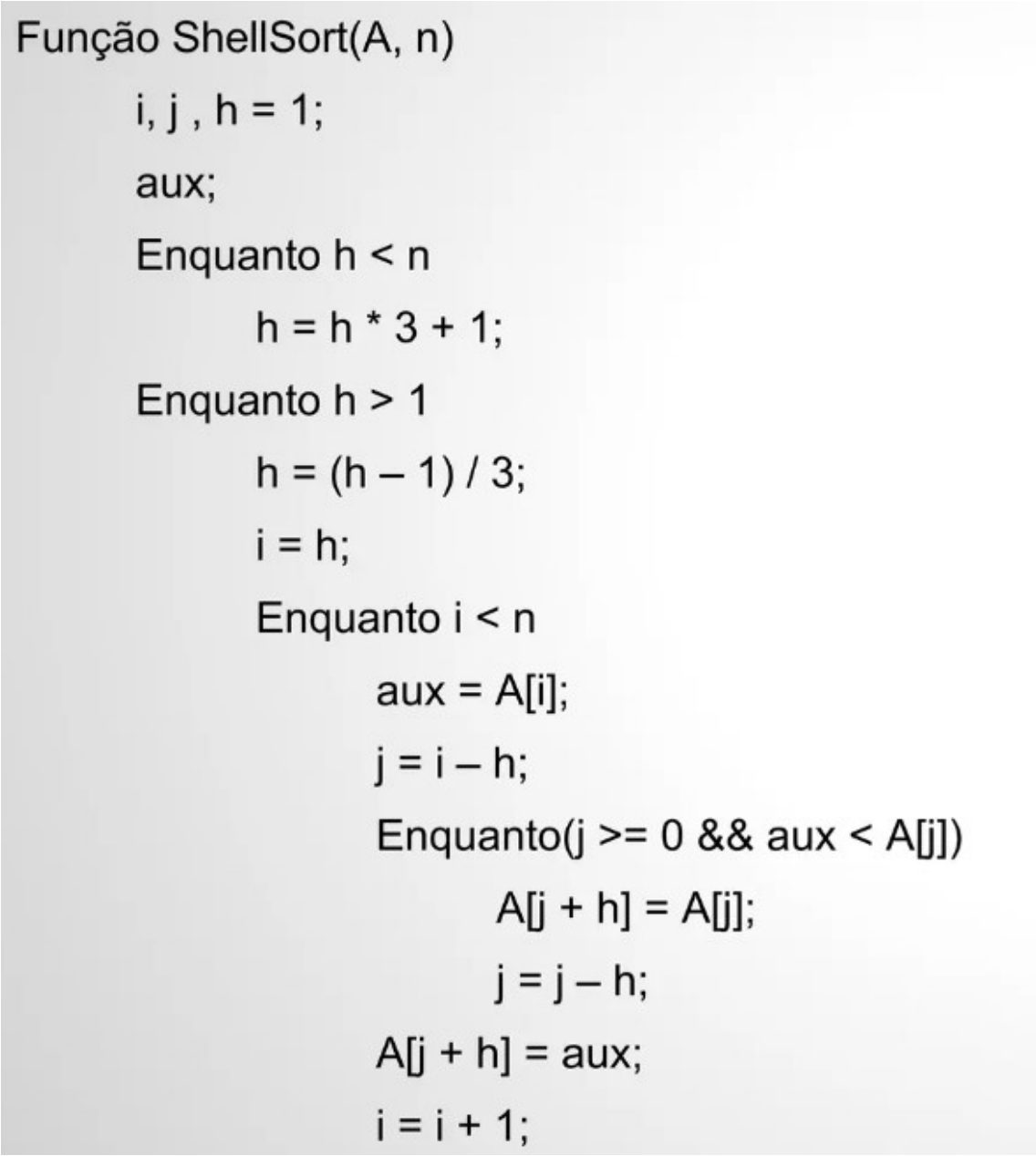

Figura 8: Pseudocodigo do algoritmo Shell Sort

6 ALGORITMO MERGE SORT

Os passos recursivos se tornaram a maneira mais eficiente de lidar com grandes arrays em que suas comparações, ações e repetições são feitas quase que infinitas, fazendo com que os arrays sejam partidos em partes menores, levando ao menor número de ações dentro do array em um só passo, assim, o array sendo ordenado mais rapidamente. Porém, a grande desvantagem destas recursões são a falta de estabilidade na execução das mesmas quando lidam com quantidade de elementos muito grande, sendo ainda assim, uma alternativa não garantida de que seu funcionamento ocorra perfeitamente.

Neste algoritmo é estudado a idéia de dividir para conquistar, onde o array é dividido em várias partes fazendo com que o escopo do array diminua facilitando a solução em partes menores, combinando elas com as partes maiores já solucionadas, facilitando também a ordenação e reformulação do array a cada passo recursivo.

Para que seja feita a diminuição do escopo, a figura 9 mostra que, no algoritmo, divide-se a sequência de n elementos ao meio, gerando duas subsequências de tamanho $\frac{n}{2}$, que por conseguinte, cada subsequência gerava mais duas subsequências gerando um passo recursivo até que cada subsequência possuísse um elemento, por fim, é feita a comparação dos elementos do escopo em que o passo recursivo possui e as trocas dos elementos deste escopo no retorno de cada passo recursivo, ordenando por fim os elementos do array.

O pseudocódigo do Merge Sort da figura 10, mostra como é fácil a implementação da recursividade no algoritmo, porém o pseudocódigo da figura 11 mostra que a implementação da combinação dos elementos necessita de três repetições para que sejam feitas as corretas combinações e ordenações dos elementos, sendo que na primeira é armazenada os elementos da esquerda do array em um array temporário e a segunda os elementos da direita do array em um outro array temporário, por fim, a terceira busca combinar por meio de comparações os elementos dos arrays temporários no array principal, onde os elementos são colocados de forma crescente.

Algumas implememtações mostram uma outra repetição na qual as outras três ocupam em ordenar os elementos em um array temporário fazendo com que a última repetição ficaria por conta de armazenar de volta todos os elementos já ordenados no array principal.

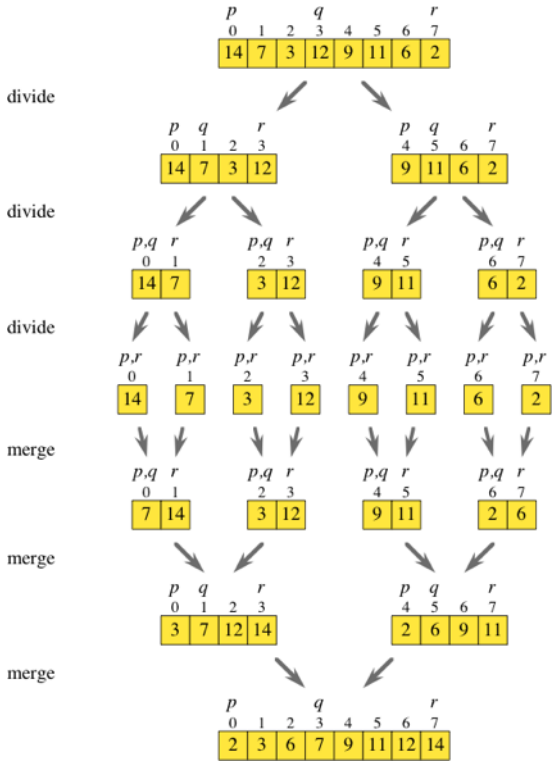

Figura 9: Exemplo de execução do algoritmo Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Figura 10: Pseudocódigo do algoritmo Merge Sort

```
MERGE(A, p, q, r)
1.  n_1 = q − p + 1
2.  n_2 = r − q
3.  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4.  for i = 1 to n_1
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n_2
7.      R[j] = A[q + j]
8.  L[n_1 + 1] = ∞
9.  R[n_2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] ≤ R[j]
14.         A[k] = L[i]
15.         i = i + 1
16.     else A[k] = R[j]
17.         j = j + 1
```

Figura 11: Pseudocódigo da função Merge

7 ALGORITMO QUICK SORT

Na necessidade de criar um algoritmo que ordenasse vetores completamente desordenados de forma que estes pudessem ser feitos mais rapidamente, utilizaram-se os recursos possíveis e conhecidos pela alta agilidade de execução na programação e na computação para gerar-se assim, um algoritmo que pudesse trazer uma resposta ordenada mais rápida que os algoritmos tradicionais.

Com isso, o Quick Sort vem com o mesmo principio que o Merge Sort para que seja feita as ordenações com menos tempo de execução, porém, como mencionado, quando o algoritmo lida com uma quantidade de elementos muito grande, provoca uma instabilidade na ordenação dos elementos do array, levando a erros inesperados de execução ou mesmo superlocação de uma pilha de memória.

A execução do algoritmo se dá pela seleção do elemento que este será o pivô, onde é particionado o array pelo elemento-pivô, como no exemplo 12. Por meio de repetições, é feitas comparações entre o atual elemento e o elemento-pivô para que seja identificada os elementos maiores e menores que o elemento-pivô, sendo assim, feitas trocas onde, os elementos maiores que o elemento-pivô vão para o lado direito do array e os menores que o elemento-pivô no lado esquerdo do array na ordem em que são selecionados. Assim, iniciando os passos recursivos em

que são selecionados pivôs dentro do escopo do passo recursivo efetuando-se as comparações e trocas, particionando-se, assim, até o último elemento.

Como o algoritmo é instável quando se trata de grande quantidade de elementos e por consequência motivo da sua ineficiência se tratando de elementos que já estão ordenados, a seleção do pivô mostra ser o fator que define sua eficiência e estabilidade na ordenação dos elementos.

O algoritmo exemplificado no pseudocódigo 13, mostra como é a implementação do Quick Sort de Lomuto, em que basicamente simplifica a implementação de Hoare, porém sendo um pouco ineficiente, mas as duas implementações são baseadas em que um pivô é selecionado no extremo do array, assim como no exemplo 12, porém as implementações vão de encontro com o problema mencionado.

Assim, para compensar a instabilidade do algoritmo, foram estudadas modificações em que buscavam trazer uma menor instabilidade na execução do algoritmo e uma melhor seleção deste pivô, gerando as versões que serão analisadas.

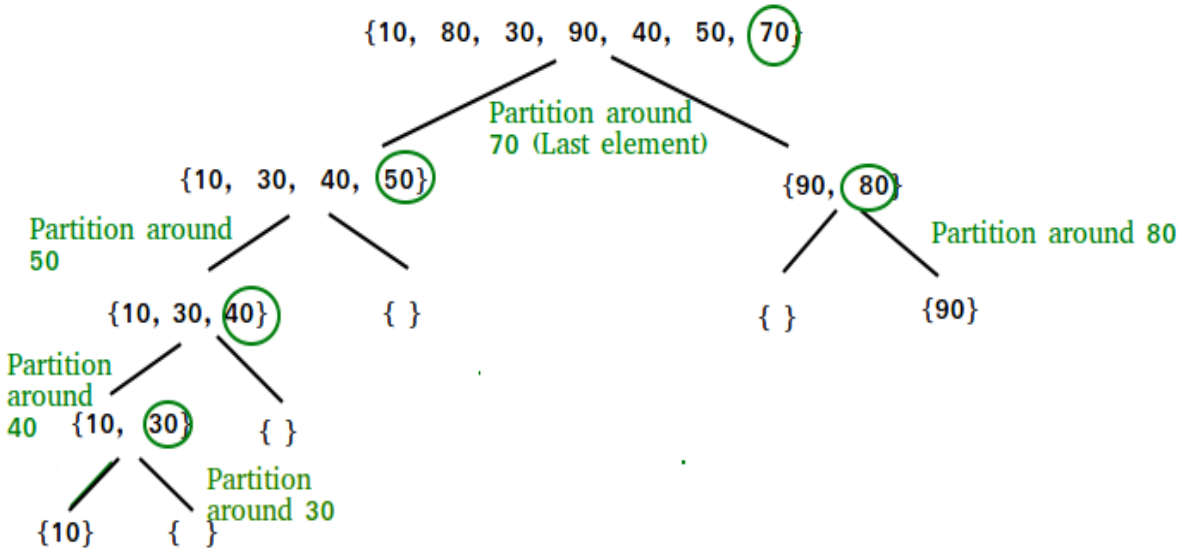

Figura 12: Exemplo de execução do algoritmo Quick Sort

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Figura 13: Pseudocódigo do algoritmo Quick Sort e a função Partição

8 ALGORITMO HEAP SORT

O algoritmo utiliza uma estrutura de dados chamada: *heap*, onde é enxergada o vetor como uma árvore binária, possuindo dois tipos de *heaps*, Heap Máximo e Heap Mínimo. O Heap Máximo coloca os vetores ordenados de forma crescente e o Heap Mínimo coloca os vetores ordenados de forma decrescente.

No *heap*, cada elemento do vetor corresponde a um nó desta árvore imaginária, assim, cada nó pode possuir um filho à direita e/ou um filho à esquerda, sendo ligadas pelo nó pai (ou raiz).

Como este vetor é enxergado como árvore, o nó-pai e os nós-filhos, possuem posições específicas no vetor, por exemplo: o nó-pai que possui posição 1 no vetor, possui os filhos da direita e da esquerda, porém, não possui irmão(outro nó ligado no mesmo nó-pai), portanto, para que seja descoberta a posição dos filhos, tanto da direita quanto da esquerda, compreende-se que cada nó-pai, tem direito a dois filhos, assim a posição pode ser denotada por $2i$, onde i = posição do nó-pai. Porém, o filho da esquerda sempre vem primeiro que o filho da direita, então, a posição $2i$, corresponde a posição do filho da esquerda, e a próxima posição ao filho da direita, assim, o filho da direita possui posição $2i + 1$.

Esta separação é feita pela função CONSTROI, onde o comprimento do vetor é cortado pela metade e o heap é construído por meio da função VERIFICA. dependendo do tipo de *heap*, busca pelo primeiro elemento encontrado que seja maior ou menor que valor do primeiro elemento do vetor(nó-raiz), efetuando-se a troca entre esses valores e, assim, efetuando outras trocas por meio da posição do elemento encontrado recursivamente até que não possua mais nós-filhos que sejam maiores ou menores que o nó-raiz.

Assim, o algoritmo Heap Sort, depende de duas funções para que seja feita a ordenação, de acordo com seu tipo. Sendo que a função CONSTROI apenas efetua a construção da árvore, colocando seus respectivos elementos, ficando para apenas a função VERIFICA, a ordenação destes elementos.

Comparado a algoritmos como Quick Sort e Merge Sort, o Heap Sort possui uma maior estabilidade, pois ele particiona o vetor em forma de árvore binária, o que leva uma menor recursividade e uma menor preenchimento de pilha de memória. Mas assim como, os algo-

ritmos que utilizam recursividade, mencionados anteriormente, como forma de ordenação, estão sujeitos à instabilidade.

9 ANÁLISE E COMPLEXIDADE DOS ALGORITMOS

Na computação entende-se que cada linha do código gera um custo computacional específico para cada operação, seja em questão de espaço(memória) ou tempo de execução, gerando cálculos de complexidade do algoritmo que compreendem entre big-O (pior caso), big-Ω (melhor caso) e big-Θ (médio caso). Para o cálculo de complexidade, é feito o cálculo da soma do produto do custo(tempo de execução) e a quantidade de vezes que cada instrução é executada, com a quantidade podendo ser multiplicada por ela mesma em mais vezes quando há mais de um laço de repetição.

9.1 Complexidade do Insertion Sort

Para o algoritmo Insertion Sort o pior caso é denotado por: $T(n) = C_1.n + C_2.(n - 1) + C_4.(n - 1) + C_5.\sum_{j=2}^{n} T_j + C_6.\sum_{j=2}^{n}(T_j - 1) + C_7.\sum_{j=2}^{n}(T_j - 1) + C_8.(n - 1)$.

Onde, $\sum_{j=2}^{n} T_j = \frac{n^2-n}{2} - 1$ e $\sum_{j=2}^{n}(T_j - 1) = \frac{n^2-n}{2}$.

Assim, a expressão resulta em: $T(n) = C_1.n + C_2.(n - 1) + C_4.(n - 1) + C_5.\frac{n^2-n}{2} - 1 + C_6.\frac{(n^2-n)}{2} + C_7.\frac{n^2-n}{2} + C_8.(n - 1)$.

Efetuando a distribuição: $T(n) = C_1 n + C_2 n - C_2 + C_4 n - C_4 + \frac{C_5 n^2 - C_5 n}{2} - C_5 + \frac{C_6 n^2 - C_6 n}{2} + \frac{C_7 n^2 - C_7 n}{2} + C_8 n - C_8$.

Fator comum em evidência: $T(n) = (\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}).n^2 + (C_1 + C_2 + C_4 + \frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2} + C_8).n + (-C_2 - C_4 - C_5 - C_8)$.

Concluindo-se que: $T(n) = \mathrm{A}n^2 + \mathrm{B}n + \mathrm{C}$.

Assim, $T(n) = O(n^2)$, pois em big-O define-se o maior valor para que seja executado, definindo-se o pior caso.

Para o melhor caso entende-se que os custos C_6 e C_7, são descartados, pois quando o vetor encontra-se ordenado, não se necessita passar pelas linhas 6 e 7 como no exemplo da Figura 2. Assim fica denotado: $T(n) = C_1.n + C_2.(n - 1) + C_4.(n - 1) + C_5.\sum_{j=2}^{n} T_j + + C_8.(n - 1)$.

Porém, $\sum_{j=2}^{n} T_j$ resultaria em $n-1$, pois não seria necessário entrar dentro do *loop*. Resultando em: $T(n) = C_1 n + C_2 n$ - $C_2 + C_4 n$ - $C_4 + C_5 n$ - $C_5 + C_8 n$ - C_8. Com o fator comum em evidência, temos: $T(n) = (C_1 + C_2 + C_4 + C_5 + C_8).n + (-C_2 - C_4 - C_5 - C_8)$. Concluindo que: $T(n) = An + B$. Assim, $T(n) = O(n)$ ou $T(n) = \Omega(n)$. Se tornando uma função linear.

9.2 Complexidade do Bubble Sort

Para o algoritmo do Bubble Sort o pior caso é compreendido em: $T(n) = C_1.(n-1) + C_2.\sum_{j=1}^{n-1} T_j + C_3.\sum_{j=1}^{n-1}(T_j - 1) + C_4.\sum_{j=1}^{n-1}(T_j - 1)$. Onde, $\sum_{j=1}^{n-1} T_j = \frac{(n^2-n)}{2}$ e $\sum_{j=1}^{n-1}(T_j - 1) = \frac{(n^2-3n)}{2} + 1$.

Assim a expressão completa fica: $T(n) = C_1 n - C_1 + \frac{C_2 n^2 - C_2 n}{2} + \frac{C_3 n^2 - 3C_3 n}{2} + C_3 + \frac{C_4 n^2 - 3C_4 n}{2} + C_4$.

Fator comum em evidência resultaria em: $T(n) = (\frac{C_2+C_3+C_4}{2}).n^2 + (C_1 - \frac{C_2-3C_3-3C_4}{2}).n + (-C_1 + C_3 + C_4)$.

Chegando a uma equação quadrática, em que: $An^2 + Bn + C$. Concluindo que $T(n) = O(n^2)$.

Agora, para o melhor caso entende-se que a linha 4 do seu pseudocodigo exemplificado na figura 4 não seria executado, resultando na expressão: $T(n) = C_1.(n-1) + C_2.\sum_{j=1}^{n-1} T_j + C_3.\sum_{j=1}^{n-1}(T_j - 1)$.

Então levando ao resultado: $T(n) = C_1 n - C_1 + \frac{C_2 n^2 - C_2 n}{2} + \frac{C_3 n^2 - 3C_3 n}{2} + C_3$.

Que formaria a expressão: $T(n) = (\frac{C_2+C_3}{2})n^2 + (C_1 - \frac{C_2-3C_3}{2})n + (-C_1 + C_3)$. Gerando um algoritmo de complexidade $T(n) = O(n^2)$, porém alguns artigos mostram uma implementação diferente da figura 4 levando a uma implementação com complexidade $O(n)$ para o melhor caso. Onde basicamente a expressão do melhor caso ficaria: $T(n) = C_1 n + C_2 n$ - $C_2 + C_4 n$ - $C_4 + C_5 n$, onde resultaria na função $An + B$ que traz complexidade linear $O(n)$.

9.3 Complexidade do Selection Sort

Para o algoritmo Selection Sort obtêm-se a seguinte expressão para definir a sua complexidade: $T(n) = C_1.(n-1) + C_2.(n-1) + C_3.\sum_{j=1}^{n-1} T_j + C_4.\sum_{j=2}^{n}(T_j - 1) + C_5.\sum_{j=2}^{n}(T_j - 1)$

$+ C_6.(n-1) + C_7.(n-1) + C_8.(n-1) + C_9.(n-1)$. Onde $\sum_{j=1}^{n-1} T_j = \frac{n^2-n}{2}$ e $\sum_{j=2}^{n}(T_j - 1) = \frac{n^2-n}{2}$.

A expressão completa seria: $T(n) = C_1n - C_1 + C_2n - C_2 + \frac{C_3n^2-C_3n}{2} + \frac{C_4n^2-C_4n}{2} + \frac{C_5n^2-C_5n}{2} + C_6n - C_6 + C_7n - C_7 + C_8n - C_8 + C_9n - C_9$.

Proferindo a equação: $T(n) = (\frac{C_3+C_4+C_5}{2})n^2 + (C_1 + C_2 - \frac{C_3-C_4-C_5}{2} + C_6 + C_7 + C_8 + C_9)n + (-C_1 - C_2 - C_6 - C_7 - C_8 - C_9)$.

Conclui-se que o algoritmo leva $O(n^2)$ para ser executado no pior caso.

O melhor caso do Selection Sort é definido pela expressão: $T(n) = C_1.(n-1) + C_2.(n-1) + C_3.\sum_{j=1}^{n-1} T_j + C_4.\sum_{j=2}^{n}(T_j - 1) + + C_6.(n-1)$. Onde $\sum_{j=1}^{n-1} T_j$ continua sendo $\frac{n^2-n}{2}$ e $\sum_{j=2}^{n}(T_j - 1) = \frac{n^2-n}{2}$.

Assim elimina-se as linhas 8, 11, 12, 13, com a expressão resultando em: $T(n) = C_1n - C_1 + C_2n - C_2 + \frac{C_3n^2-C_3n}{2} + \frac{C_4n^2-C_4n}{2} + C_6n - C_6$.

Resultando na equação: $T(n) = (\frac{C_3+C_4}{2})n^2 + (C_1 + C_2 - \frac{C_3-C_4}{2} + C_6)n + (-C_1 - C_2 - C_6)$. Concluindo também que o algoritmo leva $O(n^2)$ para ser executado também no melhor caso, pois o algoritmo entra no segundo loop, assim como exemplificado no pseudocódigo da figura x.

9.4 Complexidade do Shell Sort

O algoritmo Shell Sort não possui ao certo uma complexidade definida, visto que a partir dos testes conhecidos e expostos, o algoritmo lida de forma mais eficiente com as listas de array que o próprio Insertion Sort na qual sua implementação foi inspirada, mesmo que de acordo com sua implementação demonstre ter complexidade acima de $O(n^2)$.

9.5 Complexidade do Merge Sort

A complexidade do Merge Sort no pior caso pode ser definida por $T(n) = C_1.n + C_2.(n-1) + C_3.(n-1) + C_4.(n-1) + C_5.n + C_6.(n-1) + C_7.n + C_8.(n-1) + C_9.n + C_{10}.(n-1)$.

Com isso a expressão completa seria: $T(n) = C_1n + C_2n - C_2 + C_3n - C_3 + C_4n - C_4) + C_5n + C_6n - C_6 + C_7n + C_8n - C_8) + C_9n + C_{10}n - C_{10}$.

Resultando na expressão: $T(n) = (C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8 + C_9 +$

C_{10}).n + (-C_2 - C_3 - C_4 - C_6 - C_8 - C_{10}). Concluindo que o algoritmo leva $O(n)$ no pior caso.

Para o melhor caso, possui também a mesma complexidade que o pior caso, pois se trata de um algoritmo de ordenação recursiva, em que mesmo com os elementos ordenados, serão feitas n comparações no algoritmo. Porém o fato de ser um algoritmo recursivo, leva um tempo de $\frac{n}{2}$ para cada chamada recursiva, resultando na equação: $T(n) = T(\frac{n}{2} + T(\frac{n}{2}) + n)$, simplificando ela em: $T(n) = 2.T(\frac{n}{2}) + n)$. Assim, pelo Teorema Mestre com a função e recorrência denotada por: $T(n) = a.T(\frac{n}{b}) + f(n)$, define-se então: $a = 2$, $b = 2$, $f(n) =$ n e $n^{log_b a} = n^{log_2 2} = n$, podendo provar, assim, pelo Caso 1, em que: $T(n) = \Theta(n^{log_b a}.log_b n)$, obtêm-se então $T(n) = \Theta(n^{log_2 2}.log_2 n)$, resultando em: $T(n) = \Theta(n.log_2 n)$. Assim, conclui-se que o algoritmo para o melhor e o pior caso tem complexidade $\Theta(n \log n)$.

9.6 Complexidade do Quick Sort

O Quick Sort possui como essência a implementação do Merge Sort, porém existem diferenças-chave que, principalmente no pior caso, o desempenho do algoritmo cai em relação ao Merge Sort, elevando sua complexidade a $O(n^2)$.

Para que seja feita a análise da complexidade do algoritmo Quick Sort, devem ser considerados dois fatores: o particionamento e as chamadas recursivas.

O particionamento, vem antes das chamadas recursivas, assim, é selecionado o pivô onde este será a referência para as comparações e troca de elementos. A partir disso, entende-se que sempre há o particionamento do array antes do avanço recursivo, provocando um particionamento de n vezes do array de forma recursiva.

As chamadas recursivas dependem do pivô para que assim, seja definido até onde deve ser feitas as chamadas recursivas, efetuando partições iguais ou não.

As partições desiguais podem provocar uma sequencia de repetição de $n-1$ vezes somente para a chamada recursiva da esquerda, assim, a chamada recursiva da direita, não se procede, pois o particionamento ficou apenas para a recursão da esquerda, assim é definido por $T(n-1)$ o tempo em função da recursão da esquerda. Já que a recursão da direita é completamente descartada por conta do particionamento total do array pela recursão da esquerda, temos então $T(0)$, como representação do tempo em função da recursão da direita.

Assim, para que seja definida a complexidade, temos a expressão: $T(n) = T(n-1) +$

$T(0)+n$, onde $T(0) = 0$ e n pode ser definido por $T(n)$ ou $\Theta(n)$, resultando em um somatório de uma PA, defininda em: $T(n) = \frac{n^2+n}{2}$, por fim, trazendo uma complexidade de $O(n^2)$ configurando-se assim o pior caso para o algoritmo.

Já nas partições iguais, obtêm-se a mesma quantidade de partições $T(n)$, porém as chamadas recursivas tanto da esquerda quanto da direita são dividas igualmente, proferindo o tempo de $\frac{n}{2}$ para cada uma, resultando na expressão: $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$, que simplificando resulta em: $T(n) = 2.T(\frac{n}{2}) + n$. E assim, como na análise da complexidade do Merge Sort, é obtido uma complexidade de $\Omega(n \log n)$ para o melhor caso. Assim também se aplicaria para o médio caso, pois de acordo com Flajolet $T(n) \approx 1,386n \log n - 0,846n$, resultando também em uma complexidade $\Theta(n \log n)$ aproximadamente.

9.7 Complexidade do Heap Sort

O Heap Sort tem duas funções que onde uma delas faz a construção do heap e outra faz a ordenação, assim, na função VERIFICA, cujo, é feita a ordenação, não possui etapa de combinações assim como nos algoritmo de Quick e Merge Sort, portanto, suas operações são rápidas, com isso, o algoritmo possui tempo de execução constante, compreendendo-se que no melhor caso do algoritmo ele irá possuir complexidade $\Theta(1)$.

Já no pior caso, se o último nível tiver metade completa dos nós, entende-se que há aproximadamente $\frac{2}{3}n$ de nós. Assim, o tempo para conquistar é $T(\frac{2}{3}n)$, logo, utilizando caso 2 do Teorema Mestre é possível compreender que no pior caso, o tempo de execução do algoritmo pode chegar a $\Theta(\log_2 n)$.

Já a função CONSTROI mesmo que há um *loop*, a execução da função VERIFICA é feita $\frac{n}{2}$ vezes. Assim, mesmo no pior caso, o algoritmo consegue obter $\frac{n}{2} \log_2 n$. Assim, o algoritmo mesmo que possui, a função de construir o heap e um *loop* na qual efetua a troca entre o primeiro elemento e o ultimo elemento do heap e depois efetua a função VERIFICA, compreende-se que na função vERIFICA a quantidade nós verificados são $\log_2 n$ e também que este *loop* chega ao nó da esquerda do nó-raiz, assim, no algoritmo é feito: $\frac{n}{2}$ ($\log_2 n + n$ - 2 + $\log_2 n$). Resultando assim em: $\frac{n}{2}$ (2 $\log_2 n$ - 2). O que, deixando em evidência ficaria: $\frac{n}{2}$ 2($\log_2 n$ - 1). Aplicando, então, as operações matemáticas, resultaria em: n ($\log_2 n$ - 1), ou, aproximadamente, $n \log n$.

10 TABELAS/GRÁFICOS DOS ALGORITMOS

Nesta seção é organizado em tabelas o tempo de execução(em segundos) de cada algoritmo, a exemplificação da progressão de cada algoritmo desde de elementos menores para elementos maiores fica por conta dos gráficos expostos.

10.1 Insertion Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0150	0,0000
10000	0,0000	0,6400	0,3440
100000	0,0000	66,4220	32,2060
1000000	0,0160	6.669,6631	3.179,1111

Figura 14: Tabela com tempo de execução

Na tabela 14 é visto quanto tempo o algoritmo leva para ser executado para cada instancia e em cada caso. Que fica melhor exemplificado no gráfico 15.

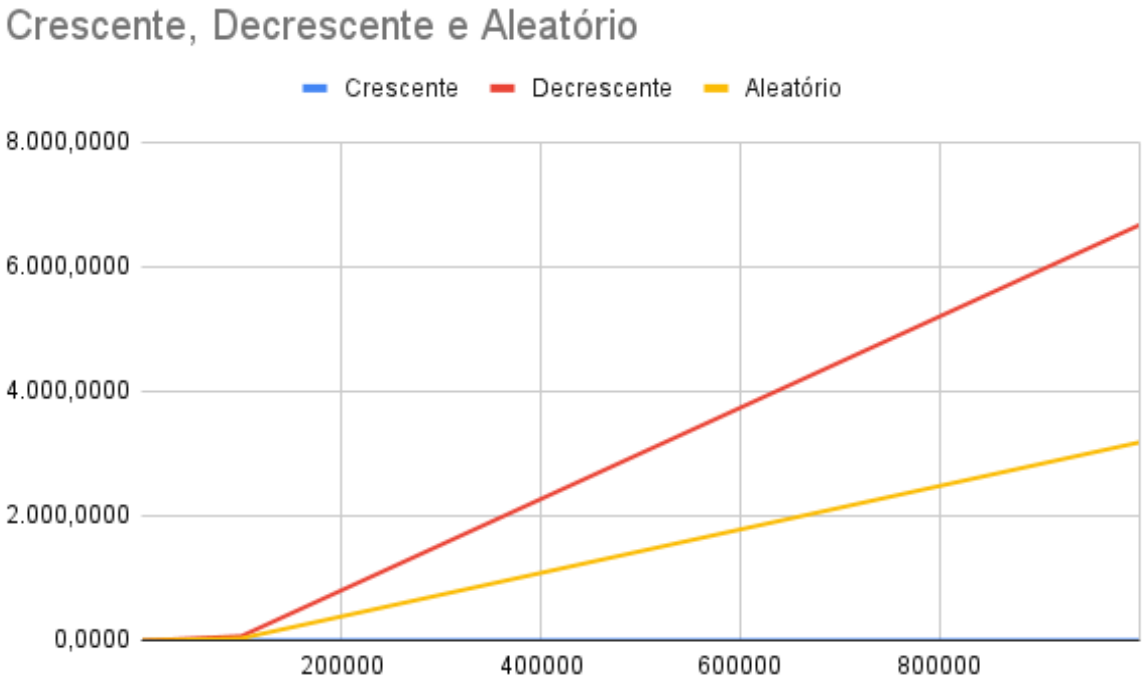

Figura 15: Gráfico de execução

10.2 Bubble Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0160	0,0310	0,0000
10000	0,4840	1,2400	0,9530
100000	49,9960	123,0370	96,1980
1000000	4.933,1670	12.348,6064	9.517,9971

Figura 16: Tabela com tempo de execução

Na tabela na figura 16 é visto quanto tempo o algoritmo levou para ser executado em cada instancia e cada caso. Que fica melhor exemplificado no gráfico da figura 17. Mostrando ser um dos algoritmos que possui o pior desempenho, no pior, melhor e médio caso. Especialmente no pior caso.

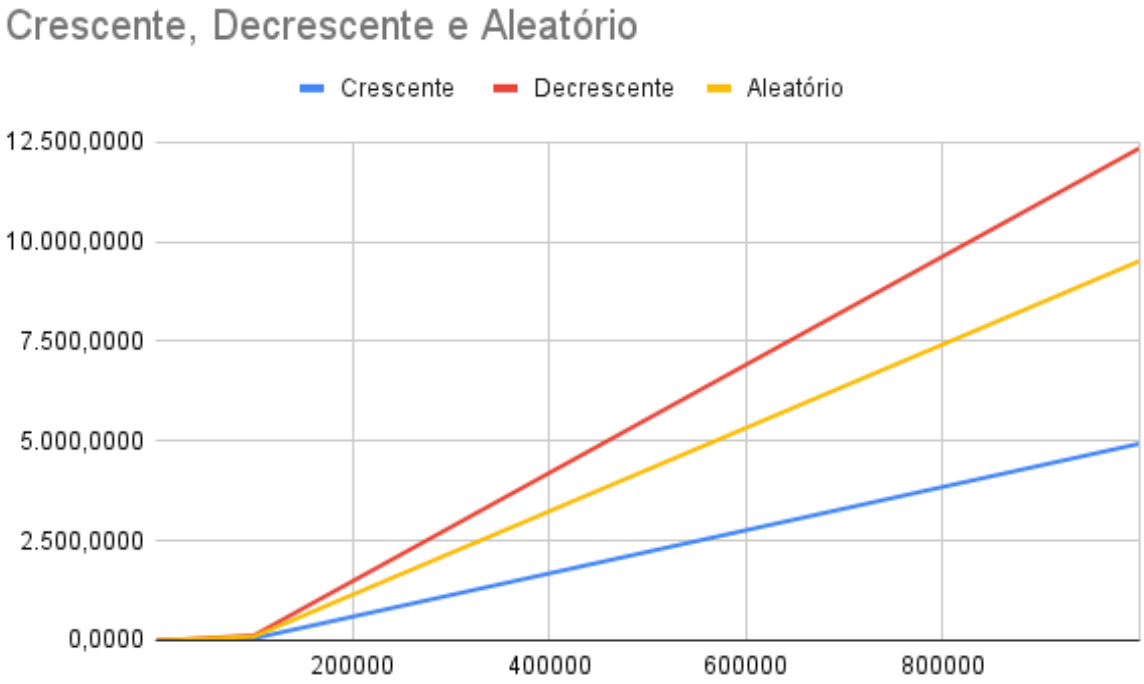

Figura 17: Gráfico de execução

10.3 Selection Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0160	0,0150
10000	0,4690	0,4840	0,5470
100000	45,9800	49,0140	53,8320
1000000	5.098,5220	4.918,3540	5.338,8369

Figura 18: Tabela com tempo de execução

É mostrado no gráfico 19 o desempenho do Selection Sort no melhor, pior e médio caso. E com a tabela 18 é concluído que sua complexidade $O(n^2)$ se aplica a todos os casos quando se trata de elementos grandes.

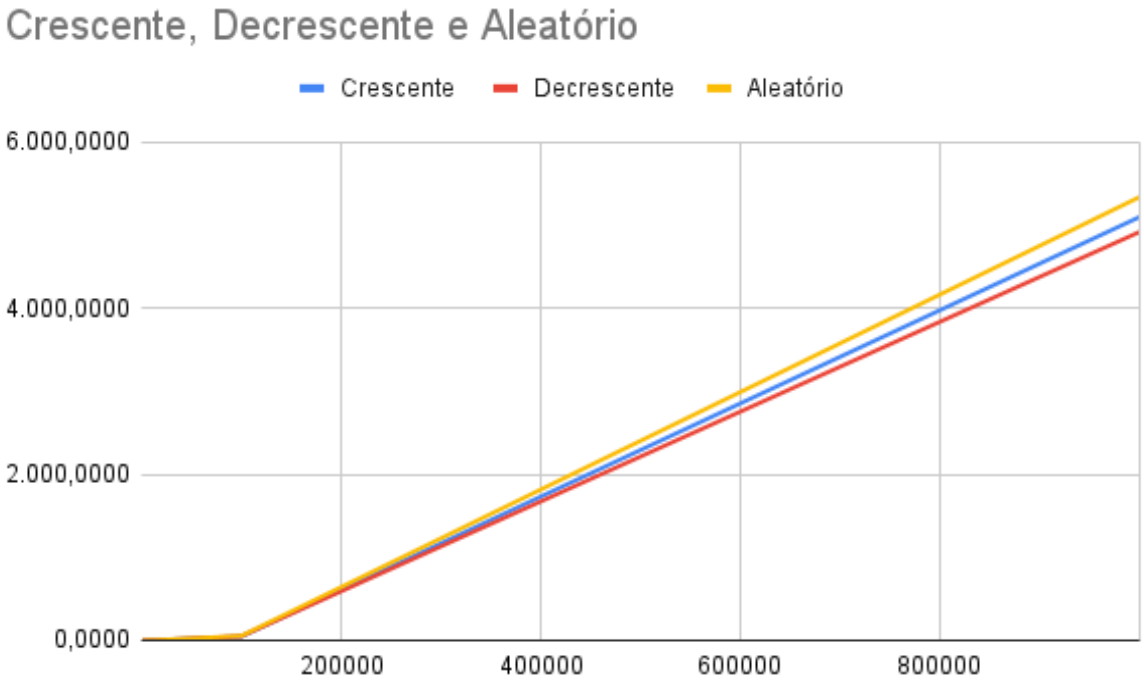

Figura 19: Gráfico de execução

10.4 Shell Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0000	0,0000
10000	0,0000	0,0000	0,0160
100000	0,0160	0,0470	0,0630
1000000	0,2350	0,2810	0,9370

Figura 20: Tabela com tempo de execução

O Shell Sort mesmo com sua essência baseada no Insertion Sort, demonstra ser um algoritmo totalmente diferente, onde, na ordenação dos elementos, é mostrado uma rapidez e fluidez que não foi visto nos algoritmos tradicionais. Na tabela 20 é visto um tempo de execução em milésimos, até mesmo em elementos enormes, onde no gráfico 21 é exemplificado melhor essa pequena diferença no desempenho do algoritmo nas instâncias menores e maiores.

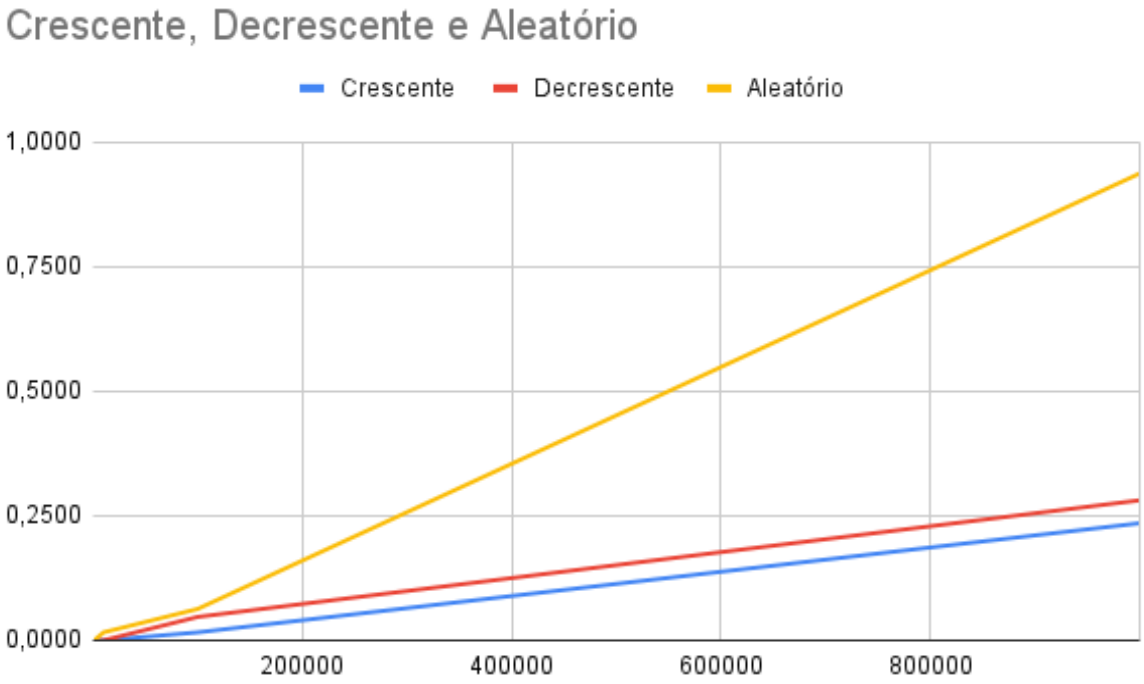

Figura 21: Gráfico de execução

10.5 Merge Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0150	0,0000	0,0000
10000	0,0780	0,0000	0,0000
100000	0,7970	0,0630	0,0620
1000000	8,5310	0,7030	0,8900

Figura 22: Tabela com tempo de execução

Já passando para os algoritmos de recursivos, o Merge Sort demonstra na tabela 22 e no gráfico 23 que houve uma queda apenas na instância de números crescentes, o que ainda sim, não retira sua complexidade menor comparada aos outros algoritmos tradicionais estudados.

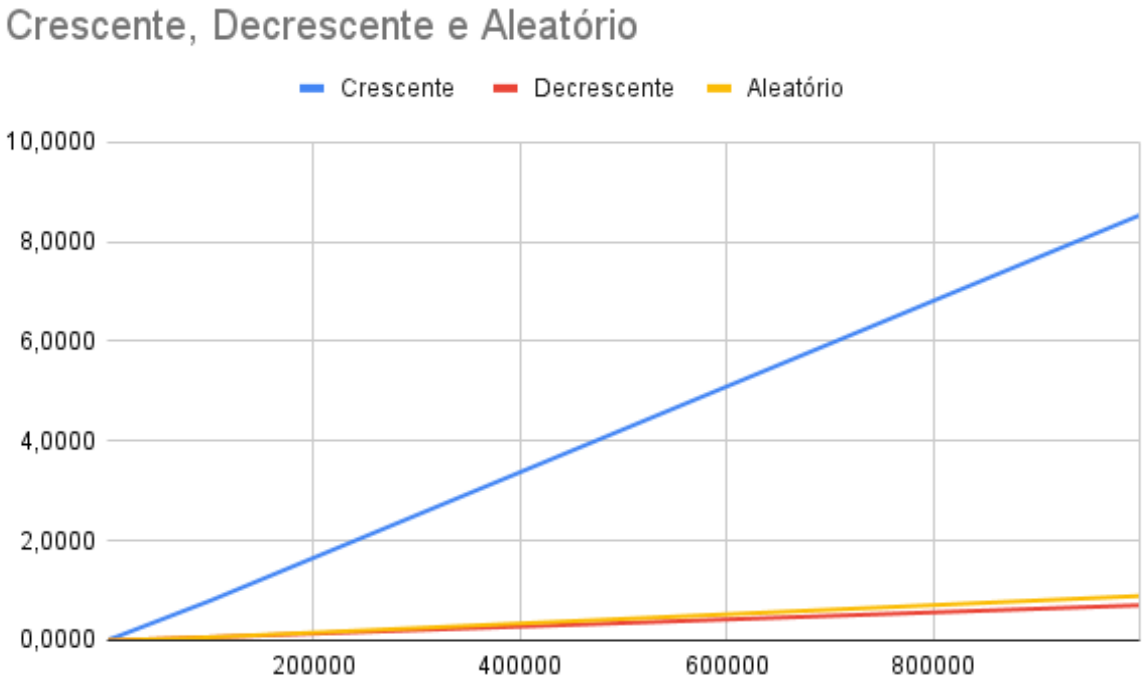

Figura 23: Gráfico de execução

10.6 Quick Sort

Foram estudados várias versões incluindo o Quick Sort tradicional e algumas com melhorias nas implementações do algoritmo, onde buscava trazer, além de maior estabilidade, um melhor desempenho na ordenação dos elementos diminuindo assim, a sua complexidade na execução da ordenação, igualando ao Merge Sort de complexidade $O(n \log n)$.

10.6.1 Versão 1

	Crescente	Decrescente	Aleatório
10	0,0000	0,0150	0,0000
100	0,0000	0,0780	0,0000
1000	0,0160	0,1880	0,0000
10000	19,8830	2,1870	0,0000
100000	-	-	0,0320
1000000	-	-	0,4530

Figura 24: Tabela com tempo de execução

Nesta versão compreende-se que sua implementação não é estável por não ser possível efetuar a ordenação no pior caso, assim como visto na tabela 24, com as instâncias 100.000 e 1.000.000 não foram possíveis suas execuções corretas e completas, deixando-se assim, indefinidas o desempenho do algoritmo no pior caso. Porém é compreendido que, como o algoritmo é recursivo, o funcionamento e estabilidade do algoritmo, depende da capacidade de pilha de cada máquina na execução. E como, esta versão é a que se originou o algoritmo, o pivô escolhido sempre permanece nos extremos, levando assim, uma recursão enorme para que a pilha pudesse comportar. Assim, como no gráfico 25 não é possível analisar em instâncias acima de 100.000.

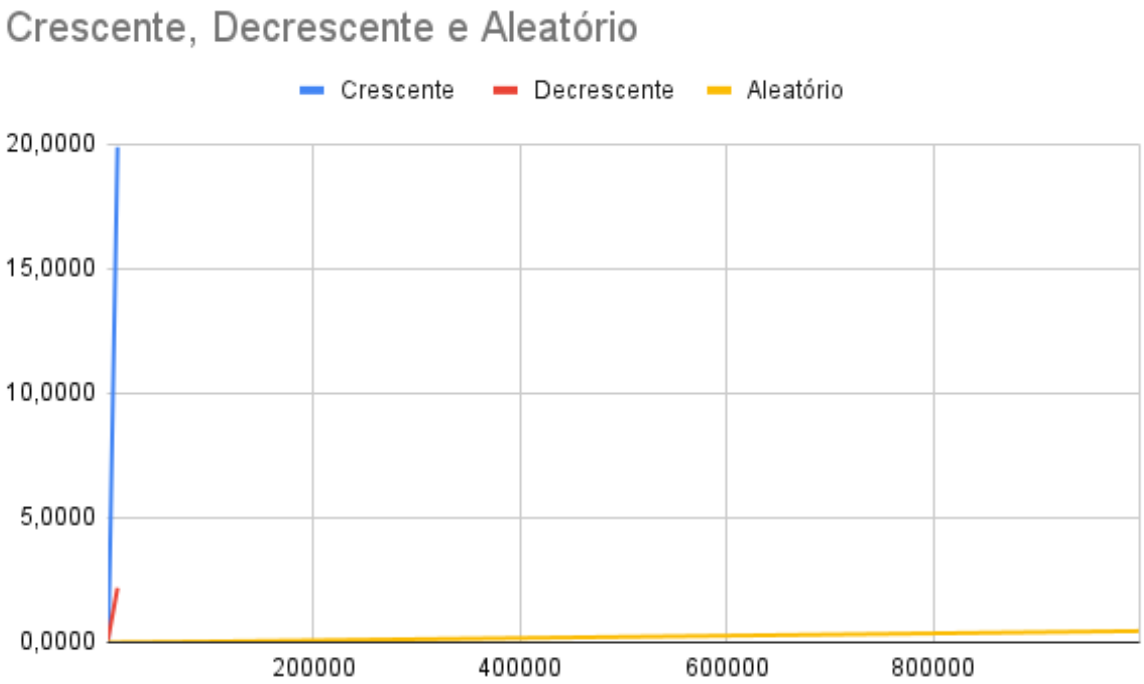

Figura 25: Gráfico de execução

10.6.2 Versão 2

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0000	0,0000
10000	0,0000	0,0160	0,0000
100000	0,0150	0,0160	0,0470
1000000	0,2180	0,2190	0,4850

Figura 26: Tabela com tempo de execução

Já nesta versão, é feito algumas mudanças na implementação do Quick Sort: o pivô é selecionado por média, fazendo com que seja utilizadas as duas chamadas recursivas presentes na implementação, divindo-se assim, as recursões enormes numa pilha só. Com isso, percebe-se uma estabilidade maior em relação a versão 1 e um desempenho superior nas instâncias Crescentes e Decrescentes(pior caso). Na tabela 26 é possível definir o tempo de execução nas maiores instâncias, sendo possível projetar no gráfico 27.

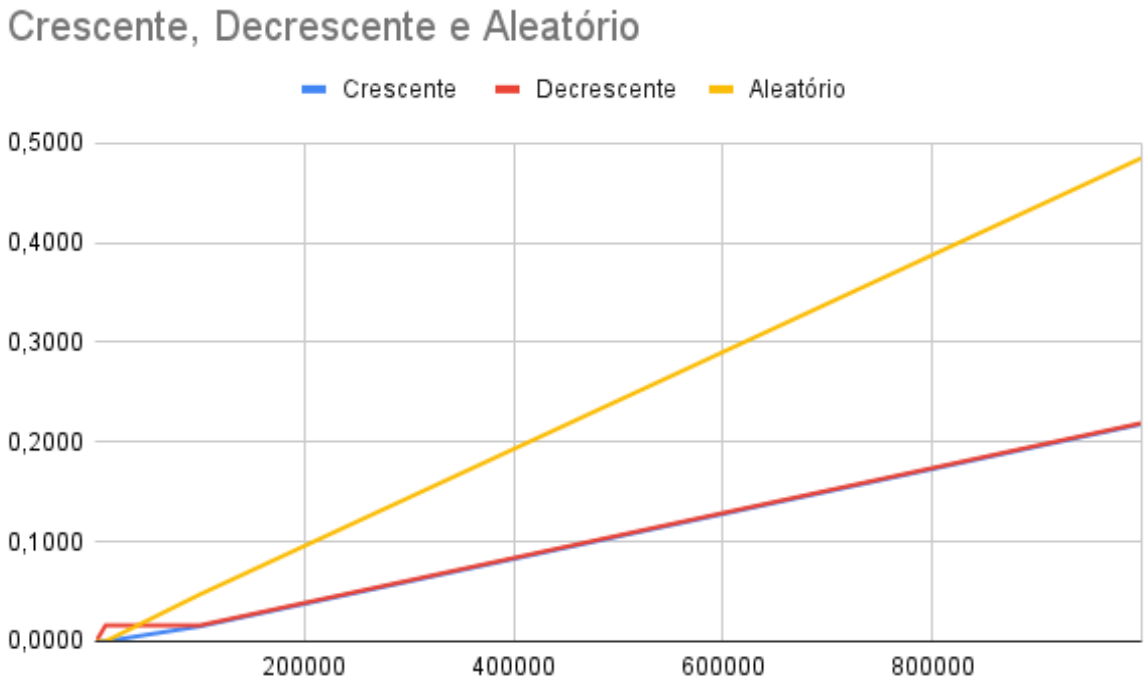

Figura 27: Gráfico de execução

10.6.3 Versão 3

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0000	0,0010
10000	0,0010	0,0020	0,0030
100000	0,0150	0,0130	0,0280
1000000	0,1400	0,1510	0,3400

Figura 28: Tabela com tempo de execução

Na versão 3, é implementado uma outra melhoria da versão 1, com o pivô sendo selecionado por mediana dos elementos, assim, desempenhando um pouco melhor que a versão anterior, mantendo um tempo de execução bastante rápido o que ainda não retira sua complexidade $O(n \log n)$, como visto na tabela 28 e sendo projetado no gráfico 29.

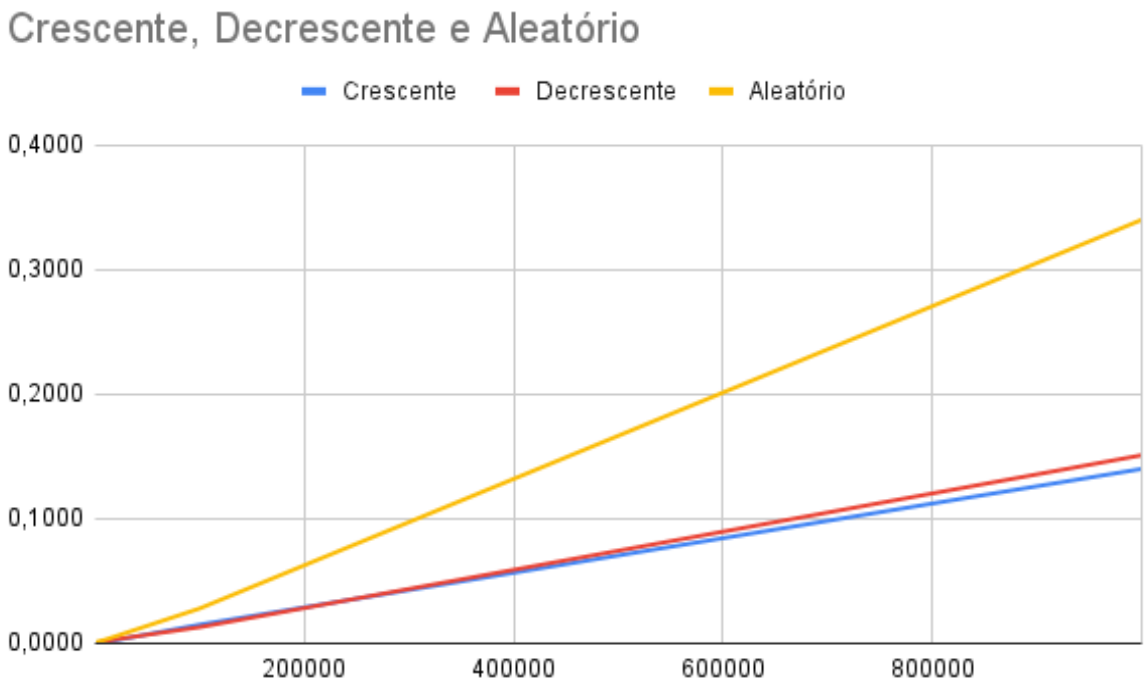

Figura 29: Gráfico de execução

10.6.4 Versão 4

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0000	0,0000	0,0000
10000	0,0160	0,0000	0,0160
100000	0,0320	0,0460	0,0470
1000000	0,4850	0,4680	0,5160

Figura 30: Tabela com tempo de execução

Nesta versão, percebe-se uma leve queda, sendo quase imperceptível na execução da ordenação, onde o pivô é selecionado de forma aleatória. O que pode levar a uma variação também do desempenho da ordenação de acordo com a posição do pivô escolhido. Assim, entende-se então que o pivô escolhido na tabela 30 foi escolhido em uma das melhores posições, projetando-se assim como no gráfico 31.

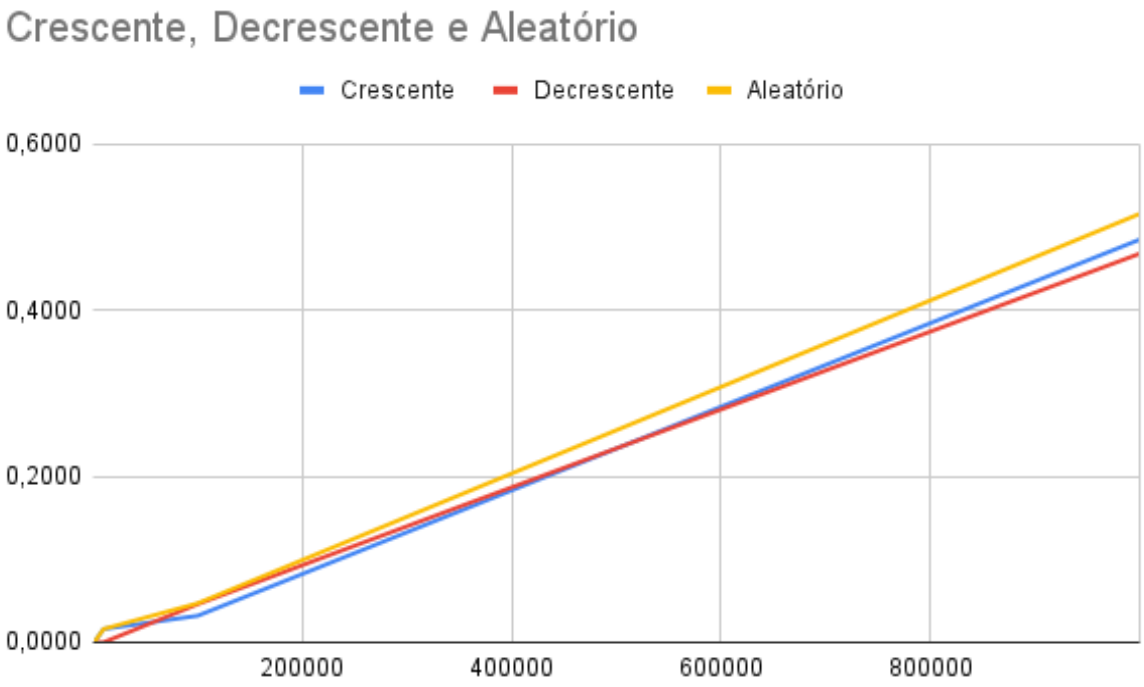

Figura 31: Gráfico de execução

10.7 Heap Sort

	Crescente	Decrescente	Aleatório
10	0,0000	0,0000	0,0000
100	0,0000	0,0000	0,0000
1000	0,0010	0,0000	0,0000
10000	0,0050	0,0040	0,0050
100000	0,0450	0,0460	0,0550
1000000	0,5310	0,5250	0,7670

Figura 32: Tabela com tempo de execução

Comparado aos outros algoritmos, o Heap Sort possui um dos melhores desempenhos em relação a ordenação dos elementos, mesmo com tantas chamadas de funções e alguns loops para ajudar na ordenação destes vetores, o que evita também uma sobrecarga maior na pilha de memória na execução das ordenações de forma recursiva. Por isso, a tabela 32 mostra quanto tempo o algoritmo leva e o gráfico 33 exemplifica melhor este desempenho.

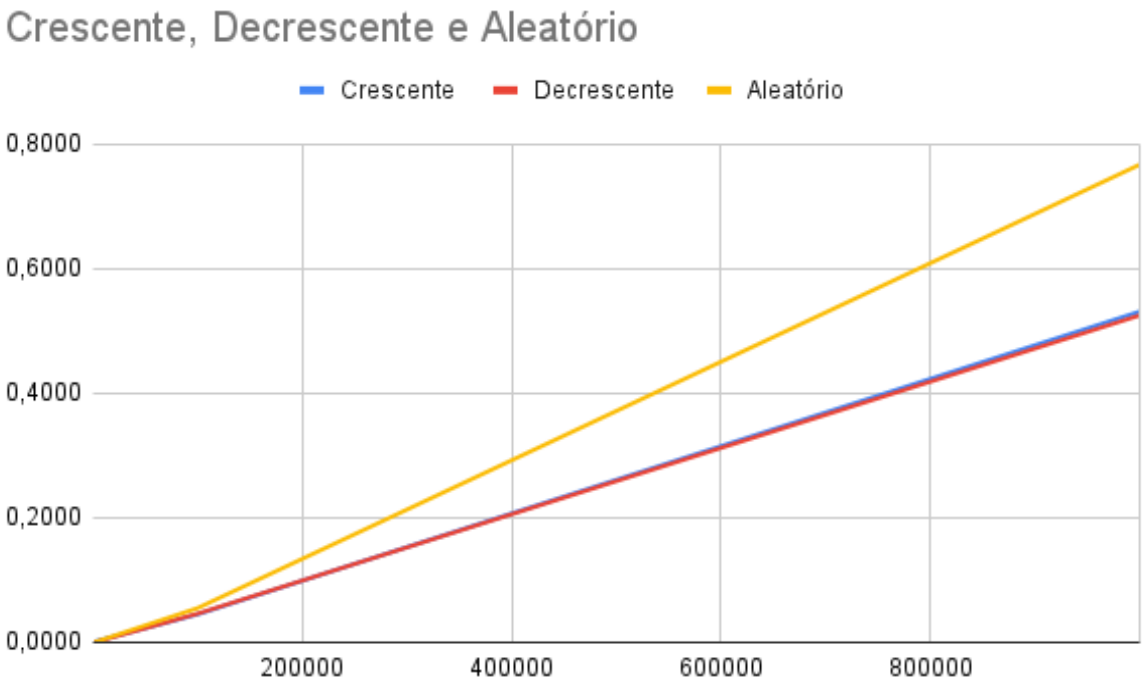

Figura 33: Gráfico de execução

11 CONCLUSÃO

Alguns algoritmos têm como um dos principais problemas a sua ineficiência em relação às ordenações por meio das implementações em que cada algoritmo é estruturado quando lidam com quantidades de elementos enormes. Porém, é compreendido que é de suma importância entender cada estrutura da computação e da programação para que seja obtida o melhor resultado e uma melhor resposta quando se trata de resoluções rápidas nas implementações de algoritmos e programas, na qual, prestam serviços para o usuário.

12 REFERÊNCIAS BIBLIOGRÁFICAS

APPIAH, O.; MARTEY, E. M. Magnetic bubble sort algorithm. International Journal of Computer Applications, Citeseer, v. 122, n. 21, 2015.

BEDER, D. M. Algoritmos de ordenação: Mergesort. BRAGAMONTE, J.; ANTUNES, F. M.; NEVES, B. S. Paralelização do algoritmo quick-sort multi-pivot com a utilização de threads. Anais do Salão Internacional de Ensino, Pesquisa e Extensão, v. 10, n. 2, 2018.

CID CARVALHO DE SOUZA. Complexidade de Algoritmo 1. 2011. Disponível em: www.ic.unicamp.br/~zanoni/teaching/mo417/2011-2s/aulas/handout/04-ordenacao.pdf. Acesso em: 10/10/2022.

FURAT, F. G. A comparative study of selection sort and insertion sort algorithms. International Research Journal of Engineering and Technology (IRJET), v. 3, n. 12, p. 326–330, 2016.

GANAPATHI, P.; CHOWDHURY, R. Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort. The Computer Journal, 2021.

GARCIA, A. M.; CERA, M. C.; MERGEN, S. L. Analisando o desempenho da paralelização no algoritmo de ordenação mergesort in-place. 13a. Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD-RS 2013), p. 123–126, 2013.

GEEKS FOR GEEKS. Insertion Sort. 2022. Disponível em:https://www.geeksforgeeks.org/insertion-sort/. Acesso em: 12/10/2022.

JOÃO ARTHUR BRUNET. Ordenação por Comparação: Insertion Sort. 2019. Disponível em: https://joaoarthurbm.github.io/eda/posts/insertion-sort/.
Acesso em: 12/10/2022.

JOÃO BRUNET. Ordenação por Comparação: Selection Sort. 2019. Disponível em: https://joaoarthurbm.github.io/eda/posts/selection-sort/. Acesso em: 20/10/2022.

KHAN ACADEMY. Analise do Selection Sort. 2017. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort. Acesso em: 26/10/2022.

LIPU, A. R. et al. Exploiting parallelism for faster implementation of bubble sort algorithm using fpga. In: IEEE. 2016 2nd International Conference on Electrical, Computer and

Telecommunication Engineering (ICECTE). [S.l.], 2016. p. 1–4.

LUIZ CHAIMOWICZ. Projeto e Análise de Algoritmos Análise de Complexidade. 2013. Disponível em: https://homepages.dcc.ufmg.br/~chaimo/paa/Aula20120-20Complexidade20-20Intro.pdf. Acesso em: 06/10/2022.

RICARDO CAMPELLO. Complexidade em Tempo de um Algoritmo, Como medir? 2009. Disponível em: https://slideplayer.com.br/amp/3661153/. Acesso em: 15/10/2022.