Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Phelipe Romano - 8135

Análise Comportamental de Algoritmos de Ordenação

Rio Paranaíba - MG

2023

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

Phelipe Romano - 8135

Análise Comportamental 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

2023

1 RESUMO

Os algoritmos de ordenação são amplamente utilizados em diversas áreas, desde a computação até a biologia. Eles são responsáveis por organizar dados de maneira eficiente e rápida, tornando possível a realização de tarefas complexas em um curto período de tempo.

Este trabalho tem como objetivo analisar os algoritmos de ordenação presentes na literatura, comparando o desempenho de cada um deles em diferentes casos e suas complexidades. Essa análise é importante para identificar qual algoritmo é mais adequado para cada tipo de problema, levando em consideração fatores como o tamanho dos dados, a complexidade do algoritmo e a eficiência na execução.

Para isso, os algoritmos *insertion sort*, *selection sort*, *bubble sort*, *shell sort*, *merge sort* e *quick sort* serão analisados e compararados os tempos de execução, com base nos conhecimentos adquiridos em sala de aula.

Sumário

2 INTRODUÇÃO

A ordenação de dados é uma tarefa comum em muitas áreas, desde a computação até a biologia. Ela é responsável por organizar informações de maneira eficiente e rápida, tornando possível a realização de tarefas complexas em um curto período de tempo. Por exemplo, ao procurar um contato na lista telefônica, seria difícil encontrar o nome desejado se os nomes não estivessem em ordem alfabética. É fácil perceber que as atividades que envolvem algum método de ordenação estão muito presentes na computação.

No entanto, qual método utilizar? Geralmente, a resposta é bem simples: podemos fazer manualmente, definindo os critérios de seleção e ordenando por si mesmo. Isso funciona bem, mas dependendo do que se deseja, pode ser cansativo e, em alguns casos, até inviável realizar a ordenação desta forma. Por exemplo, se fosse necessário ordenar vários números em ordem decrescente, 10 ou 20 seria fácil, mas e se fossem 1.000.000 números? Simplesmente não valeria a pena o esforço, não atualmente.

Sabendo disso, o método manual não é muito interessante, mas definir o melhor método também não é tão simples. Neste caso, é necessário realizar uma análise para avaliar qual seria o melhor, sendo esse o objetivo deste trabalho. Ao longo da leitura, serão apresentados e explicados diversos algoritmos de ordenação, de modo a entender seu funcionamento teórico de forma simples. Será apresentado o algoritmo de cada método em C++ e como ele opera, e, por fim, uma avaliação de cada método, contendo dados e argumentos que vão resultar em uma conclusão.

3 ALGORITMOS

3.1 *Insertion Sort*

O *insertion sort* é um algoritimo de ordenação por inserção que recebe como entrada um vetor de números e uma variavel contendo seu tamanho, retornando como saída este mesmo vetor com seus elementos ordenados de forma crescente. O algoritmo pode ser visto na função da figura 1.

```cpp
void insertion_sort(int vetor[], int tamanho){
    int j, chave, i;

    for(j = 1; j < tamanho; j++){
        chave = vetor[j];
        i = j - 1;
        while(i >= 0 && vetor[i] > chave){
            vetor[i+1] = vetor[i];
            i--;
        }
        vetor[i+1] = chave;
    }
}
```

Figura 1: Código do *insertion sort* em C++

Assim que a função é chamada e o vetor e o seu tamanho foram inseridos, são criadas três variaveis do tipo inteiro:

j - recebe o indice do elemento que vai ser ordenado.

i - contem o indice dos elementos a esquerda do elemento de indice j.

chave - armazena o valor do elemento de indice j.

Após a declaração das variaveis é utilizado um for para percorrer todos os elementos do vetor a partir da posição de indice 1, a variavel chave recebe o valor de um elemento enquanto i recebe o indice do elemento a sua esquerda. Em seguida é iniciado um while, que verifica se é o fim do vetor e se algum elemento a esquerda é maior que o valor em chave. Caso as duas condições sejam verdadeiras o elemento a esquerda passa para a direita, caso alguma delas seja falsa o elemento a direita do atual elemento de indice i recebe o valor de chave. Desta forma o for percorre os elementos do vetor enquanto o while compara e ordena os números.

Para simplificar a compreensão podemos utilizar como exemplo a ordenação das cartas de um baralho no Tabletop Simulator, conforme a figura 2.

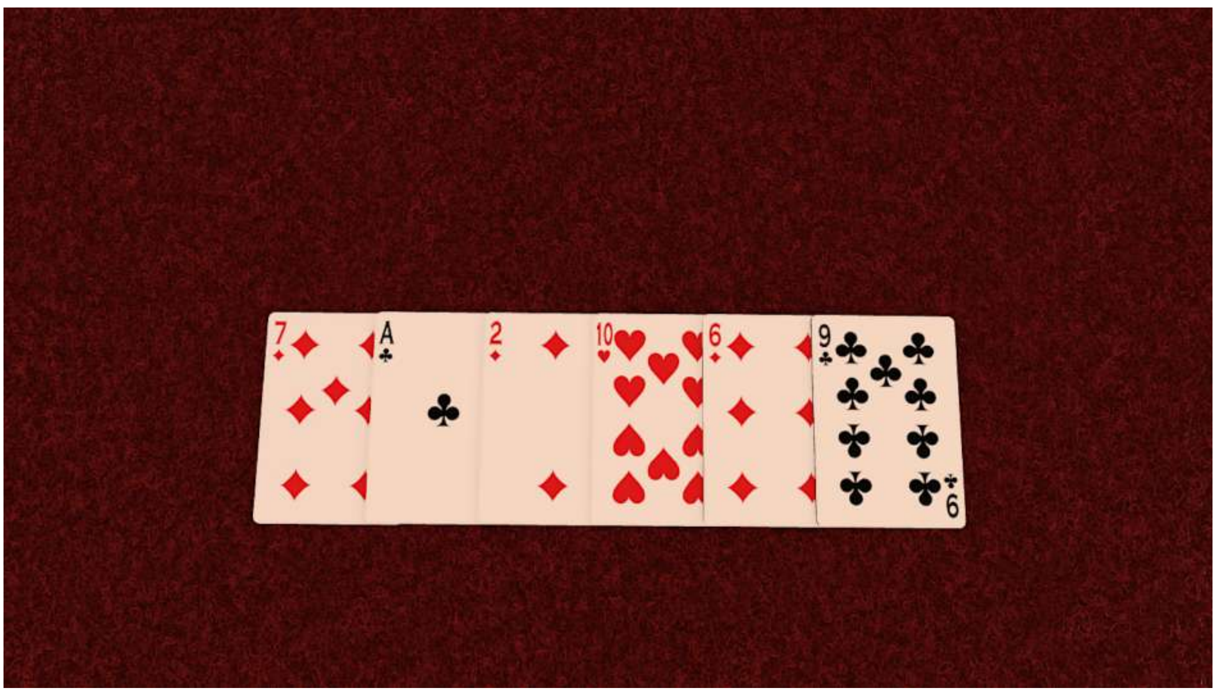
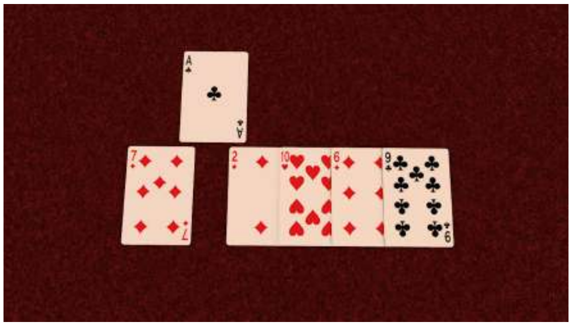
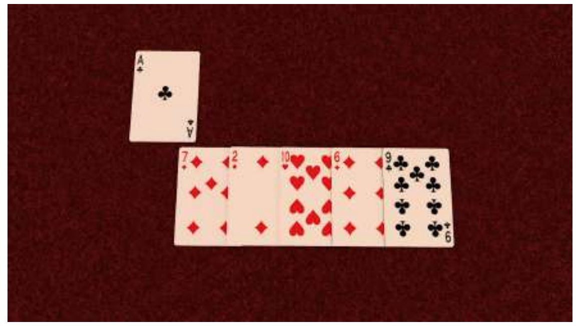


Figura 2: Conjunto de cartas não ordenado

Neste caso temos um conjunto de 6 cartas que vai representar um vetor inserido no nosso algoritimo, onde cada carta é um elemento que vamos ordenar de forma crescente. Inicialmente pegamos a carta que representa o indice 1, que nesse caso é o ás, conforme a figura 3 (a).



(a) Seleção do ás



(b) Ordenação entre o ás e o 7

Figura 3: Inicio da execução do algoritmo

Em seguida devemos comparar o ás com todos os elementos da esquerda, neste caso apenas uma carta, como o 7 é maior que o ás(carta 1 do baralho) ele vai para a posição a direita, como na figura 3 (b).

Como o ás chegou na primeira posição do vetor não temos mais como compará-lo a nenhum elemento a esquerda, logo ele é adicionado a esta posição, figura 4.

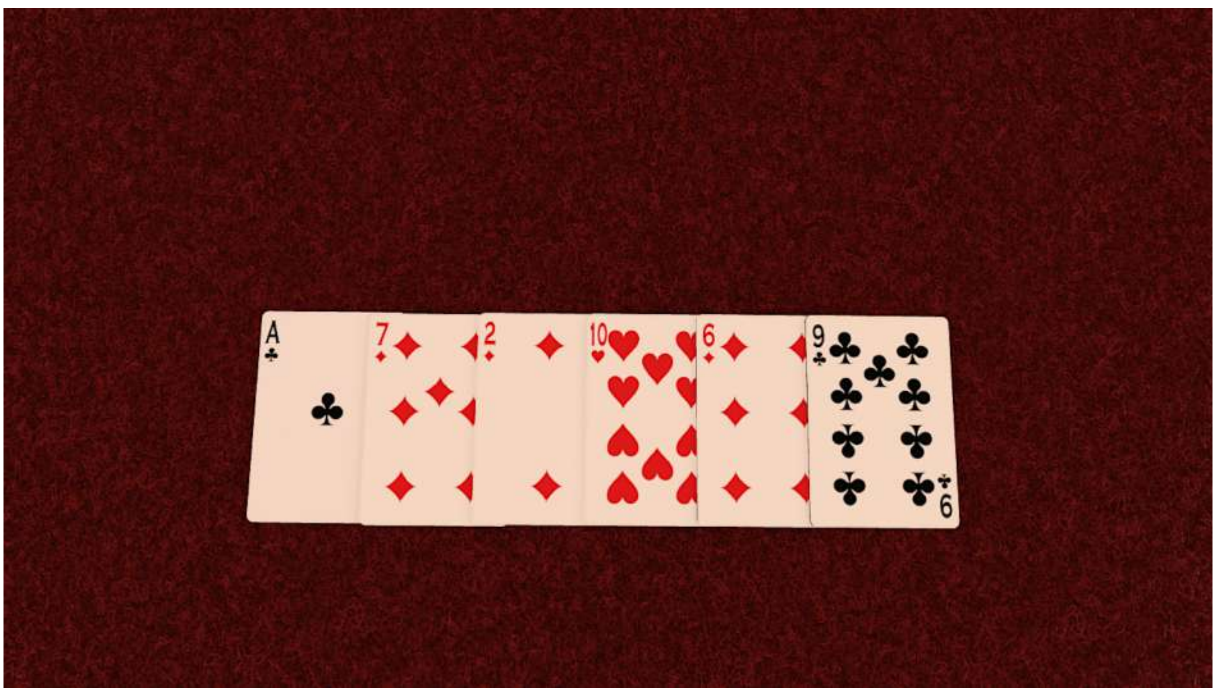
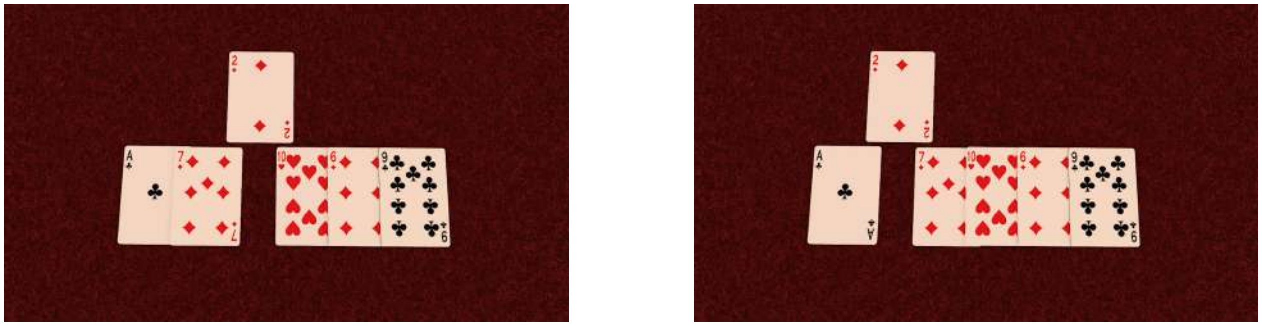


Figura 4: Os dois primeiros elementos estão ordenados

Agora devemos pegar a carta da proxima posição, neste caso o 2 para compararmos com os elementos da esquerda, ás e 7, figura 5 (a).



(a) Seleção da carta 2

(b) Ordenação entre o 2 e o 7

Figura 5: Ordenação do elemento de indice 2

Realizando mais uma comparação como na figura 5 (b) é possivel observar que o 2 não é menor que o ás, logo ele deve ficar nesta posição, figura 6.

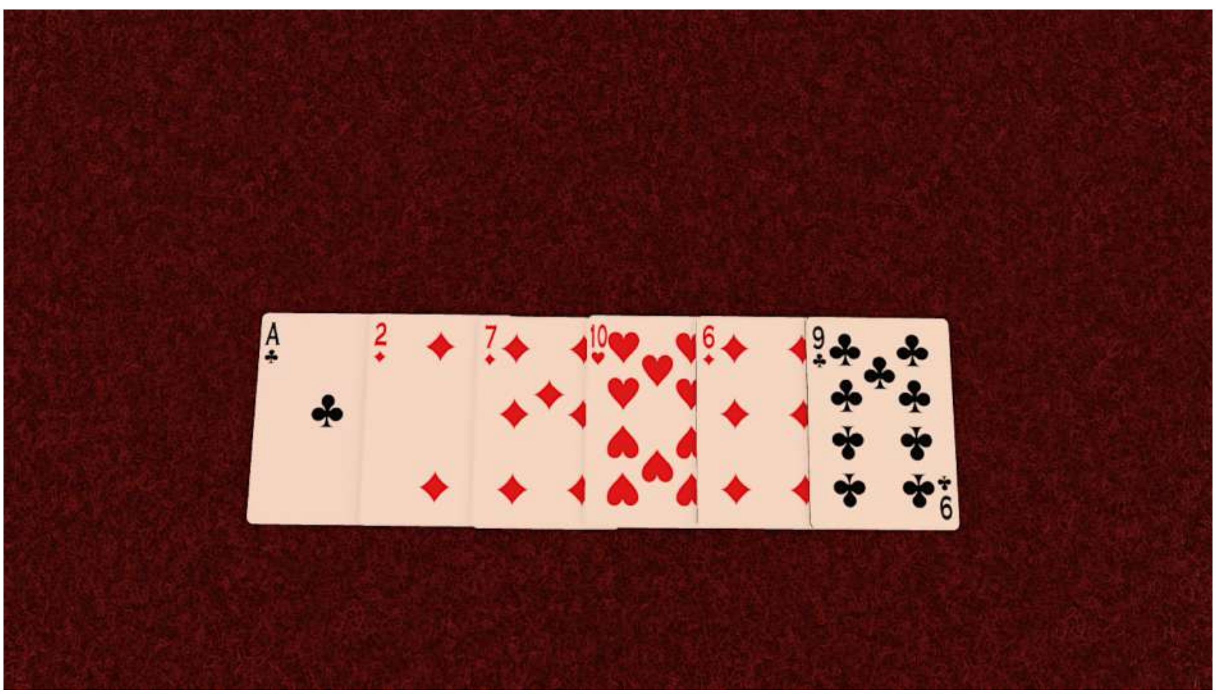
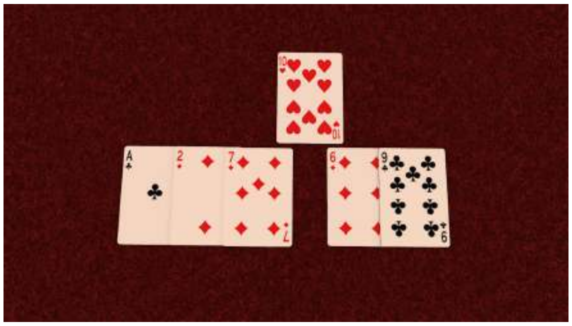
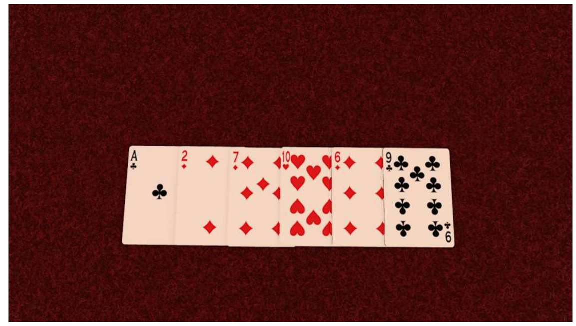


Figura 6: Os três primeiros elementos estão ordenados

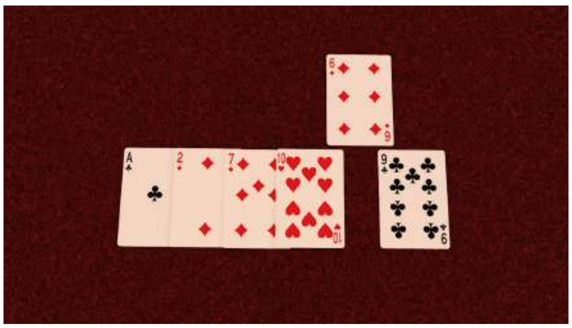
Seguindo esse mesmo algoritimo podemos ordenar o restante das cartas facilmente conforme as figuras 6 e 7.



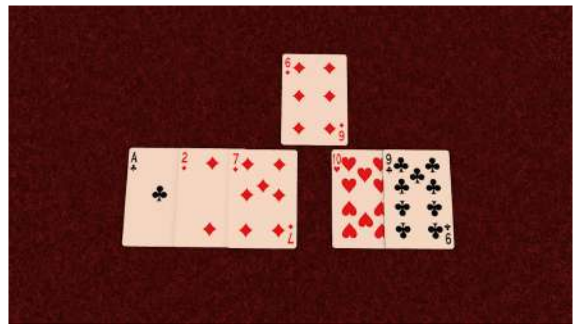
(a) Seleção do 10



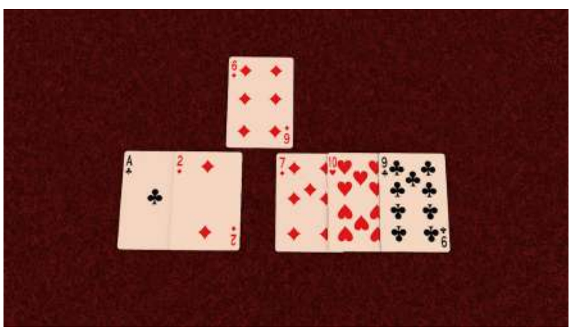
(b) Elemento ordenado



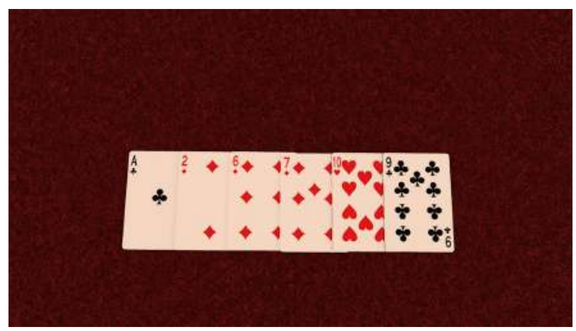
(c) Seleção do 6



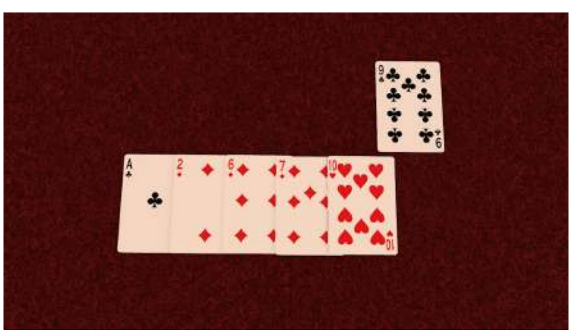
(d) Ordenação entre o 6 e o 10



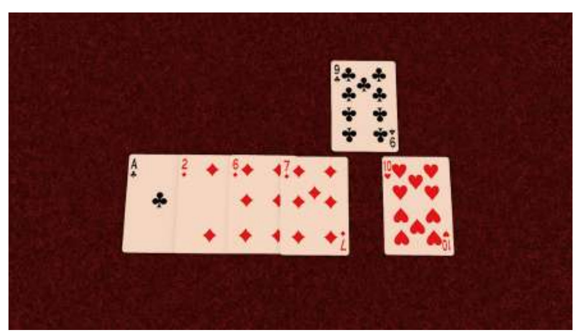
(e) Ordenação entre o 6 e o 7



(f) Elemento ordenado



(g) Seleção do 9



(h) Ordenação entre o 9 e o 10

Figura 7: Ordenação das cartas

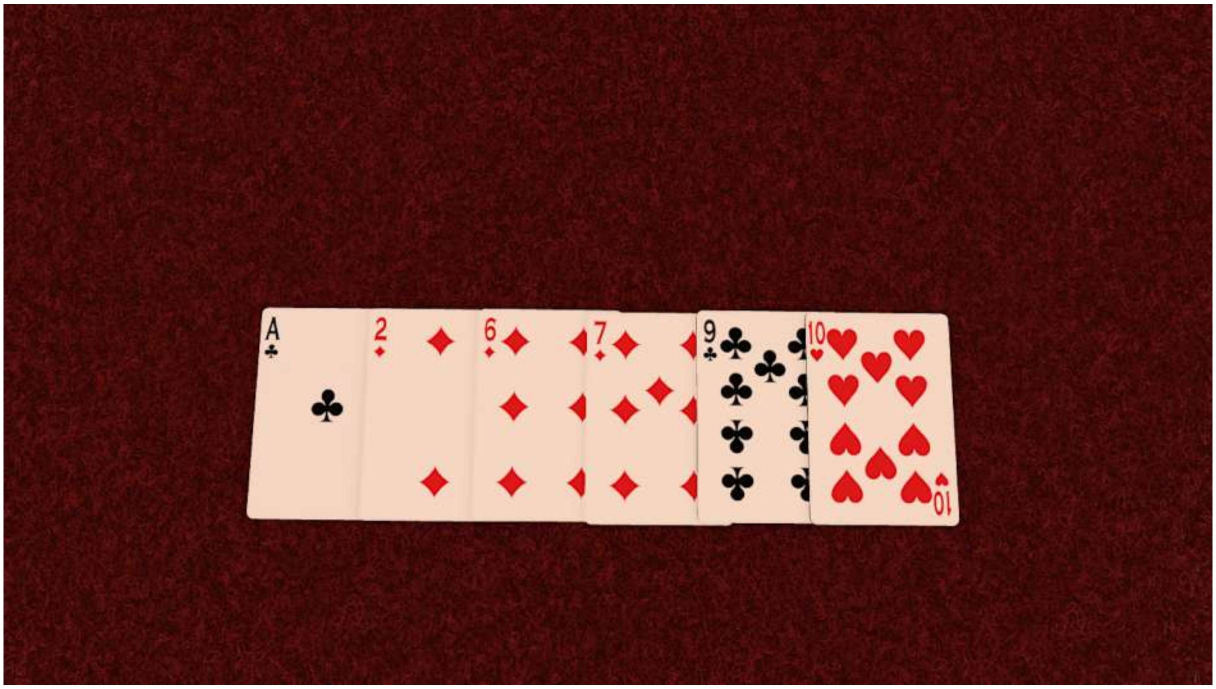


Figura 8: Todos os elementos estão ordenados

3.2 *Bubble Sort*

O *bubble sort* é um algoritmo de ordenação que a partir de um vetor contendo números de inteiros realiza diversas movimentações para ordena-los de forma crescente. O *bubble sort* possui esse nome porque os numeros são ordenados de forma a lembrar bolhas subindo até a superficie da água, onde a superficie seria a posição adequada para cada elemento. O algoritmo pode ser visualizado na função da figura 9.

```cpp
void bubble_sort(int vetor[], int tamanho){
    int chave;

    for(int i = 1; i < (tamanho - 1); i++){
        for(int j = 0; j < (tamanho - i); j++){
            if(vetor[j] > vetor[j + 1]){
                chave = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = chave;
            }
        }
    }
}
```

Figura 9: Código do *bubble sort* em C++

Esse algoritmo percorre o vetor varias vezes enquanto faz comparações entre seus elementos, e dependendo de qual for maior é realizada uma troca de posições. Além do vetor e de seu tamanho que são inseridos como parametros temos:

i - indice usado para percorrer o vetor do segundo ao penúltimo elemento.

j - indice que percorre o vetor da primeira até a posição tamanho - i.

chave - variavel auxiliar para a troca de posições.

Assim que a função é chamada a variavel chave é declarada e se dá inicio ao For responsavel por percorrer o vetor, dentro deste, outro For percorre o vetor para fazer as ordenações desde a primeira posição até o elemento de indice correspondente ao tamanho do vetor menos o valor atual de i. Cada vez que o segundo For é executado o If, que compara se o conteúdo correspondente ao indice j é maior que o próximo elemento, caso seja ambos os valores trocam de lugar e o programa procede repetindo esse ciclo até que i chegue no último elemento, momento este que o vetor está ordenado.

Esse algoritmo pode ser mostrado utilizando no proximo exemplo, onde utilizando o *bubble sort* 5 números são ordenados de forma crescente.

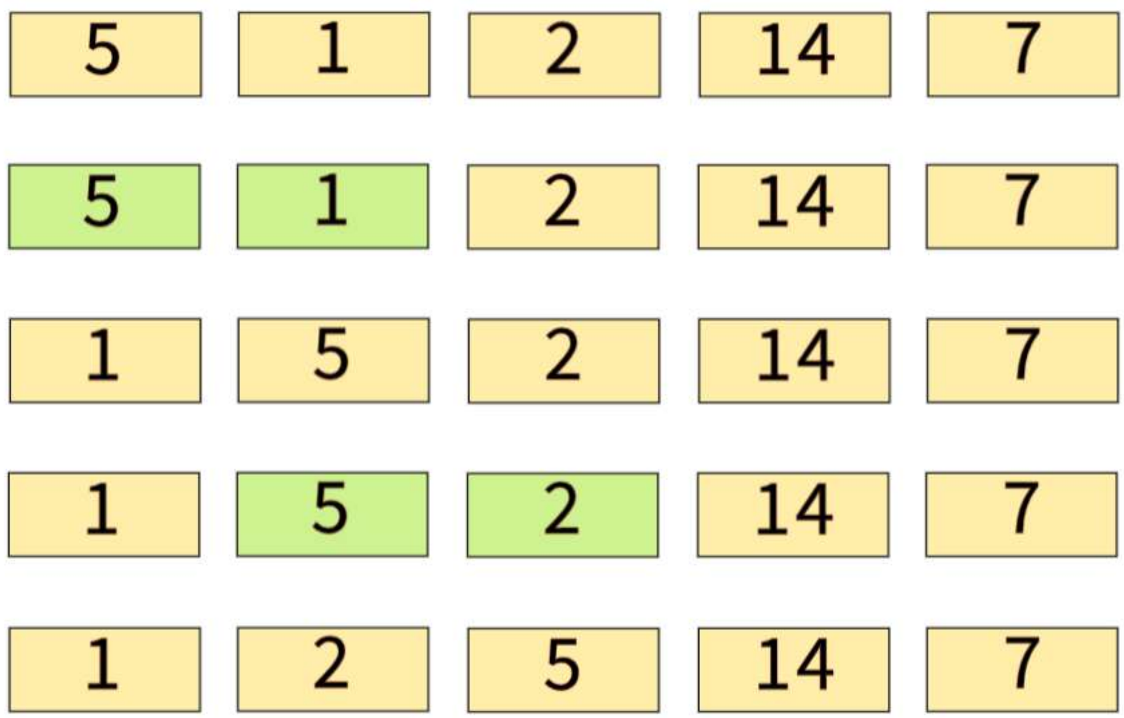


Figura 10: Inicio da ordenação de 5 elementos usando *bubble sort*

A ordenação do vetor começa selecionando os dois primeiros elementos e realizando a comparação para verificar se 5 é maior que 1, como isso é verdade os dois valores trocam de posição. Em seguida partimos para a comparação do segundo elemento com o terceiro, e como 5 é maior que 2 novamente é realizada uma troca de posições, como representado na figura 10.

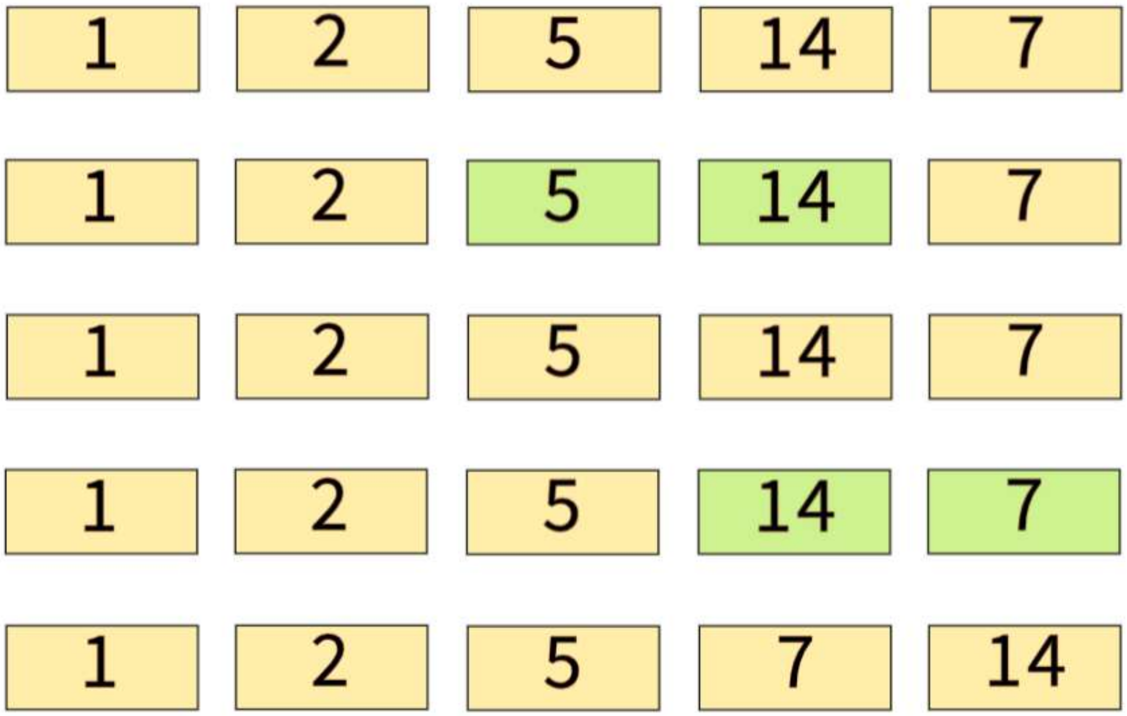
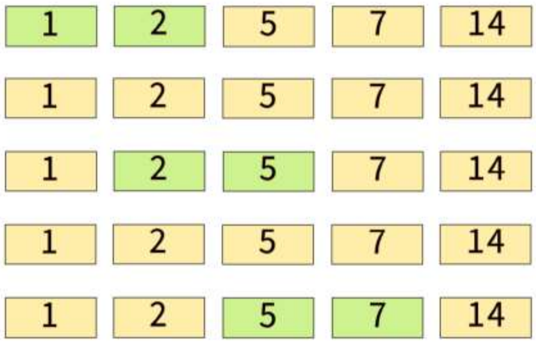
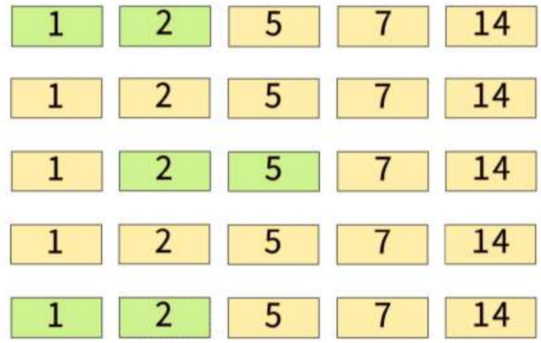


Figura 11: Ordenação de 5 elementos usando *bubble sort*

Desta vez, ao realizar a comparação se 5 é maior que 14 vemos que isso é falso, logo as posições continuam as mesmas, o mesmo não acontece com o 14 e o 7, então a posição deles é trocada. Assim a primeira execução do For foi realizada e os elementos ja estão ordenados, porem, o programa continua a execução até a condição de parada do primeiro For, que neste caso é i = 4.



(a) Segunda execução do For



(b) Terceira e quarta execução do For

Figura 12: Ordenação de 5 elementos usando *bubble sort*

Na segunda execução do primeiro For o indice do i é 2, logo o segundo For percorrera apenas até o terceiro elemento como na figura 12 (a). Como o mesmo se aplica a terceira e quarta execuções o segundo For sera executado até o segundo e primeiro elemento respectivamente representado na figura 12 (b).

3.3 *Selection Sort*

O *selection sort* é um algoritmo que assim como os outros ordena de forma crescente os elementos de um vetor de inteiros. Esse algoritmo realiza essa ordenação através de um sistema de seleção, onde o menor elemento do vetor é selecionado e colocado na primeira posição do vetor, em seguida é feito o mesmo com o segundo menor, que é adicionado a segunda posição e assim por diante. O algoritmo pode ser visualizado na figura **??**

```cpp
void selection_sort(int vetor[], int tamanho){
    int aux, chave;

    for(int i = 0; i < tamanho; i++){
        aux = i;
        for(int j = (i + 1); j < tamanho; j++){
            if(vetor[j] < vetor[aux]){
                aux = j;
            }
        }
        chave = vetor[i];
        vetor[i] = vetor[aux];
        vetor[aux] = chave;
    }
}
```

Figura 13: Código do *selection sort* em C++

Além do vetor de entrada e seu tamanho o algoritmo possui 4 outras variaveis:

i - indice usado para percorrer o vetor e achar o elemento correspondente a posição de indice i.

j - indice que percorre o vetor para procurar um elemento menor que o correspondente a posição de indice i.

chave - variavel auxiliar para a troca de posições.

aux - variavel responsavel por armazenar o indice do menor número.

No inicio da execução variaveis chave e aux são criadas e é iniciado um For que percorrera todos os elementos do vetor a partir da posição 0, dentro deste, o aux armazenara o valor de i e outro For é executado, novamente percorrendo todos os elementos do vetor a partir da posição i+1 com o objetivo de encontrar um elemento menor que o de posição aux usando um If. Assim que o For percorre todo o vetor, caso tenha encontrado ou não algum elemento menor que o de indice aux então aux recebe seu indice e então ele troca de lugar com o elemento de indice j.

Para uma melhor compreensão sera feito um exemplo onde por meio deste algoritmo deve-se organizar um conjunto de dados de forma crescente, conforme a figura 14

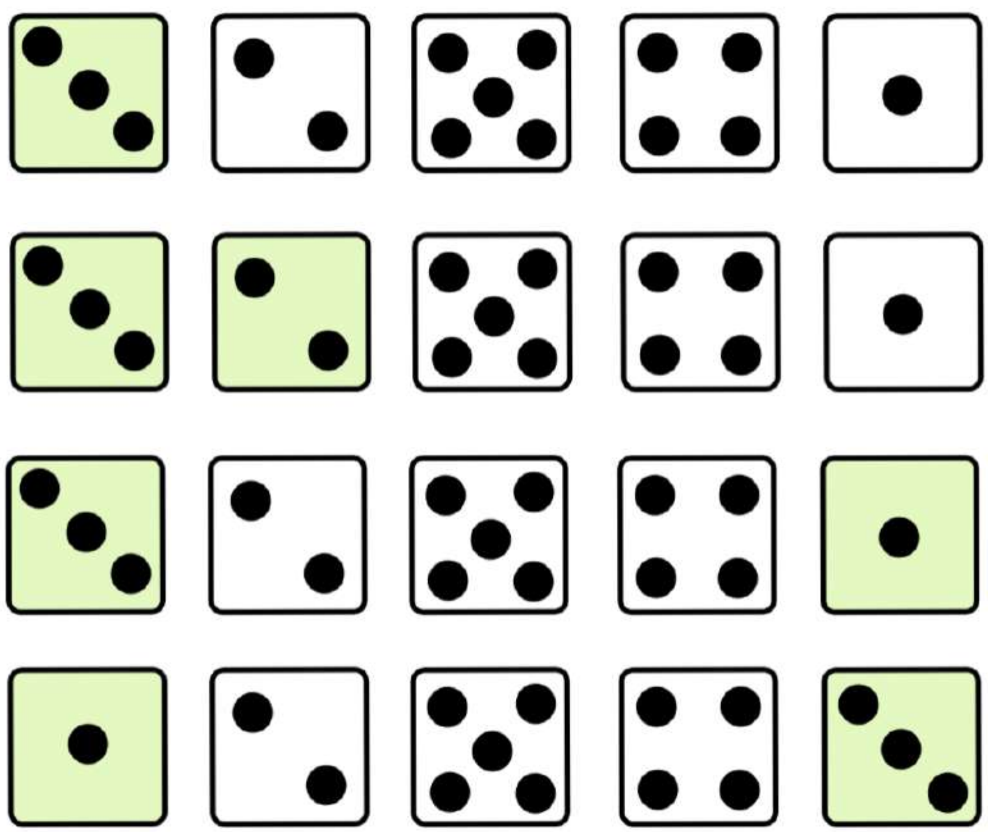


Figura 14: Código do *selection sort* em C++

Neste primeiro passo o i possui valor 1 e consequentemente aux também, então seria como se estivessemos selecionando o dado 3. Em seguida percorremos o vetor em busca de um número menor que 3, neste caso o 2 é menor, logo aux recebe seu indice, agora buscamos um que seja menor que 2, o 1 ao final do conjunto de dados é menor então aux recebe seu indice. Como o já selecionamos o menor elemento do conjunto agora realizamos a troca de posições entre o elemento de indice j e o elemento de indice aux.

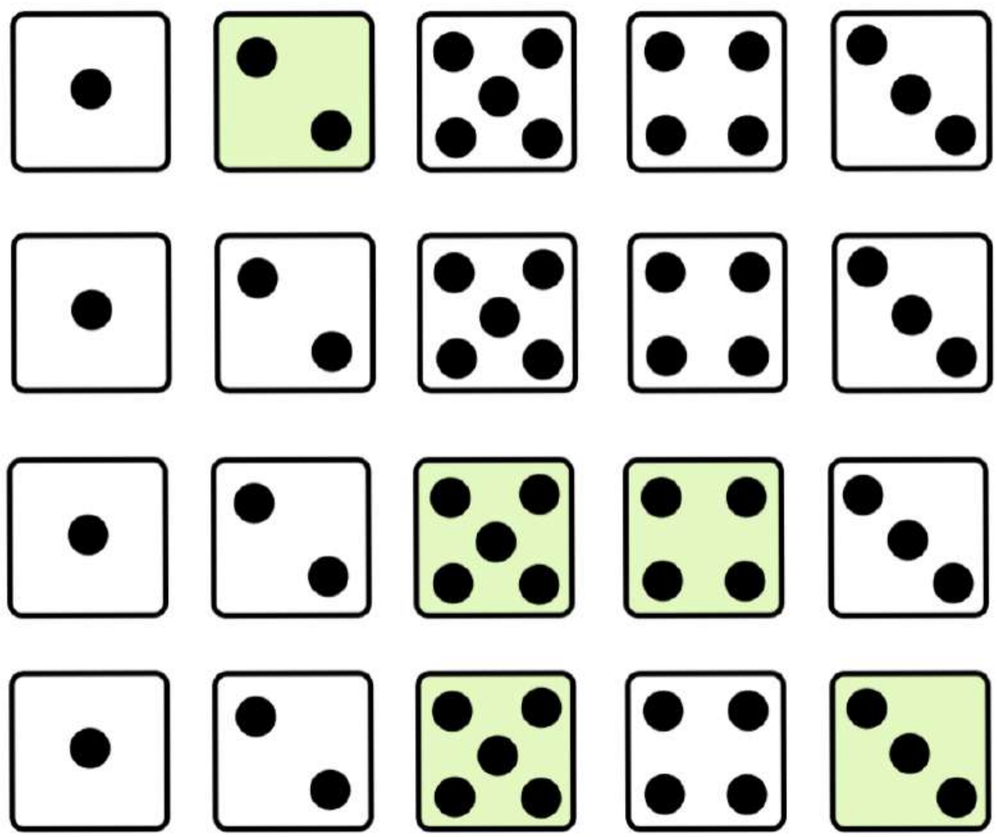


Figura 15: Código do *insertion sort* em C++

Agora que o i possui indice 1 fazemos a comparação para ver se tem algum número menor que 2, como não existe nenhum ele está na posição certa. Em seguida, com i igual a 2 realizamos a comparação do 5 com o 4 e o 3, como 3 é o menor número os dois trocam de lugar.

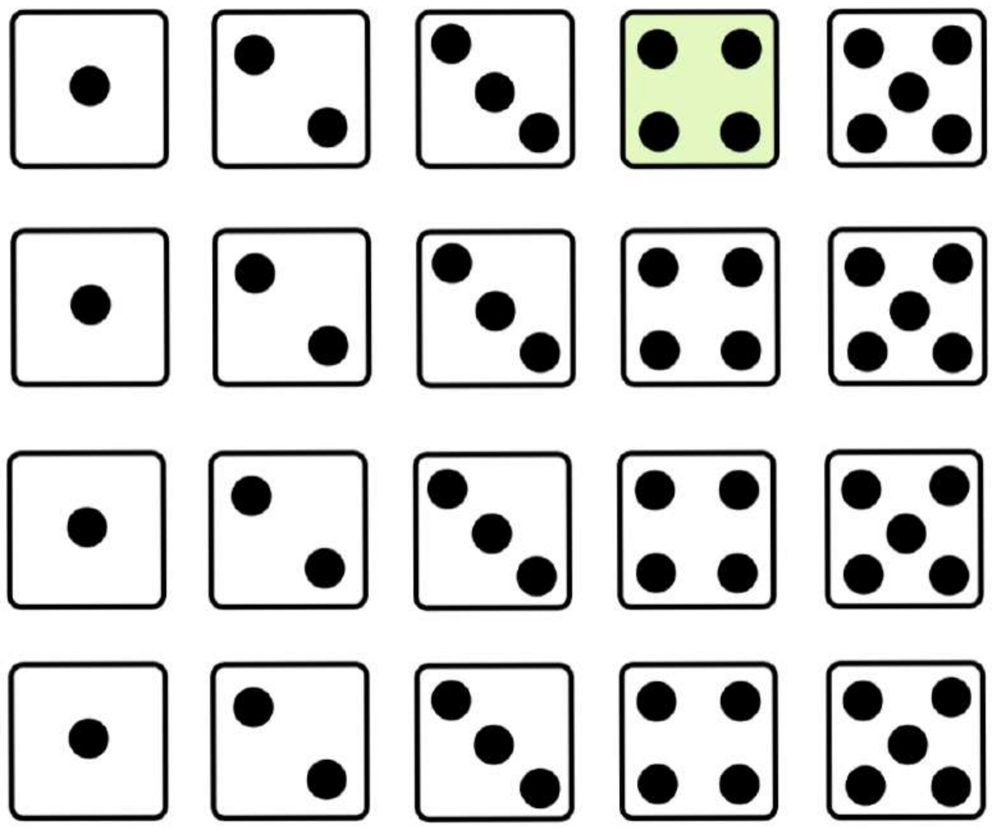


Figura 16: Código do *insertion sort* em C++

Seguindo com o algoritmo falta apenas realizar a comparação dos dois ultimos elementos, como 4 não é maior que 5 significa que ambos estão na posição certa, terminando assim a ordenação.

3.4 *Shell Sort*

O *shell sort* é um algoritmo de ordenação conhecido pela sua eficiencia, sendo capaz de ordenar um vetor de números em um tempo bem inferior em relação aos outros metódos. Esse algoritmo se baseia em fazer trocas de posição baseadas em intervalos que vão desde a metade do tamanho do vetor até ele ser menor que 1, sempre sendo dividido por 2 a cada execução, o codigo do *shell sort* pode ser visto na figura 17.

```cpp
void shell_sort(int vetor[], int tamanho){
    for (int gap = tamanho/2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < tamanho; i ++)
        {
            int temp = vetor[i];
            int j;
            for (j = i; j >= gap && vetor[j - gap] > temp; j -= gap)
                vetor[j] = vetor[j - gap];
            vetor[j] = temp;
        }
    }
}
```

Figura 17: Código do *shell sort* em C++

Quando essa função é executada inicialmente entramos em um For onde sera declarado o gap, essa variavel possui metade do valor do vetor e a cada execução deste For ele é dividido por 2 até o gap ser inferior a 1. Destro dele existe outro For que usa o i para percorrer o vetor desde o gap até o tamanho final do vetor e a cada execução aumenta em 1, em seu interior a variavel temp é declarada e ela vai pegar o valor do vetor de indice i para que dentro do proximo For, que faz a troca se posições entre o número de indice j que é declarado sendo inicialmente igual a i e a cada loop é subtraido dele o gap e o elemento de indice j-gap. Caso o j seja menor ou igual ao gap e o elemento j-gap seja maior que o valor em temp então terminamos esse primeiro ciclo. Esse sistema continua até que o vetor esteja ordenado e pode ser visualizado melhor no próximo exemplo da figura 18, onde devemos organizar a playlist do Spotfy em relação aos minutos.

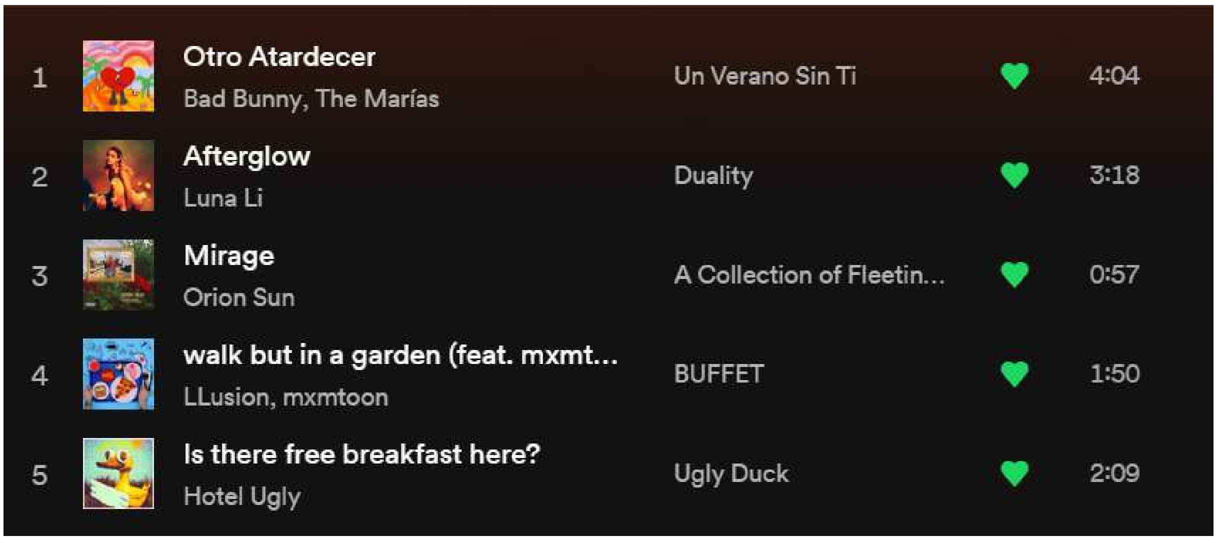


Figura 18: Playlist do Autor no Spotfy

No caso deste exemplo o gap será 2, o que significa que faremos as comparações com intervalos de 2 posições, como na figura 19.

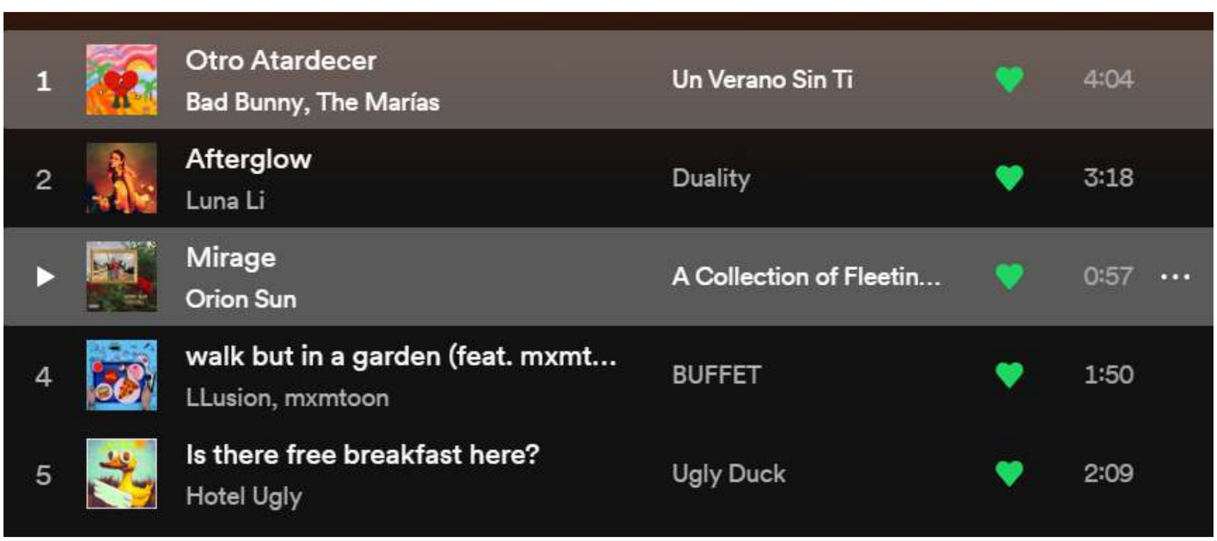
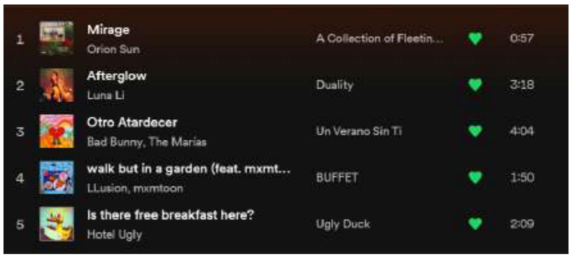
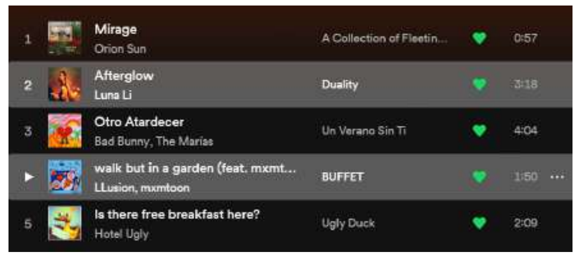


Figura 19: Ordenação da playlist

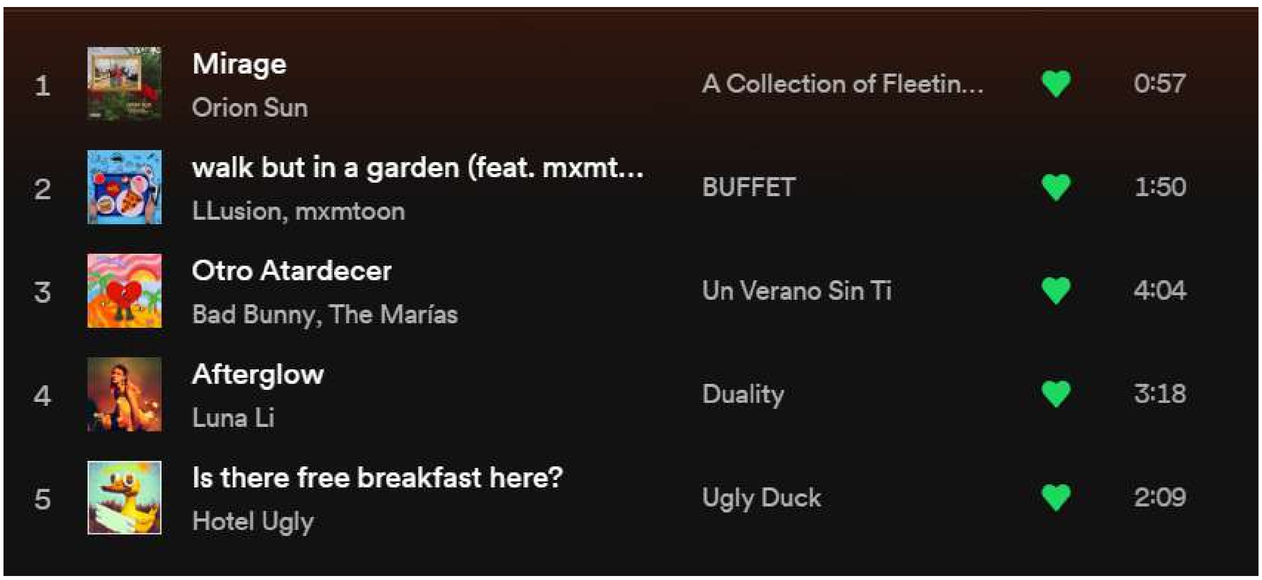
Agora selecionamos o primeiro e terceiro elementos e comparamos seus minutos, como 0 é menor que 4 vamos trocar as duas de lugar, como na figura 20 (a). Após isso podemos selecionar o proximo elemento, que no caso é o 2, e compara-lo ao elemento 4, novamente realizaremos a troca, já que 1 é menor que 3, como representado na figura 20 (a).



(a) Primeiro e terceiro elementos ordenados



(b) seleção do segundo e quarto elemento



(c) Segundo e quarto elemento ordenados

Figura 20: Ordenação dos elementos

Vamos novamente repetir essas comparações até chegarmos ao fim do conjunto, nesse exemplo o fim é neste proximo passo, então comparando agora o terceiro e quinto elementos, 4 e 2 respectivamente, vemos que como 2 é menor que 4 trocamos suas posições novamente, figura 21.

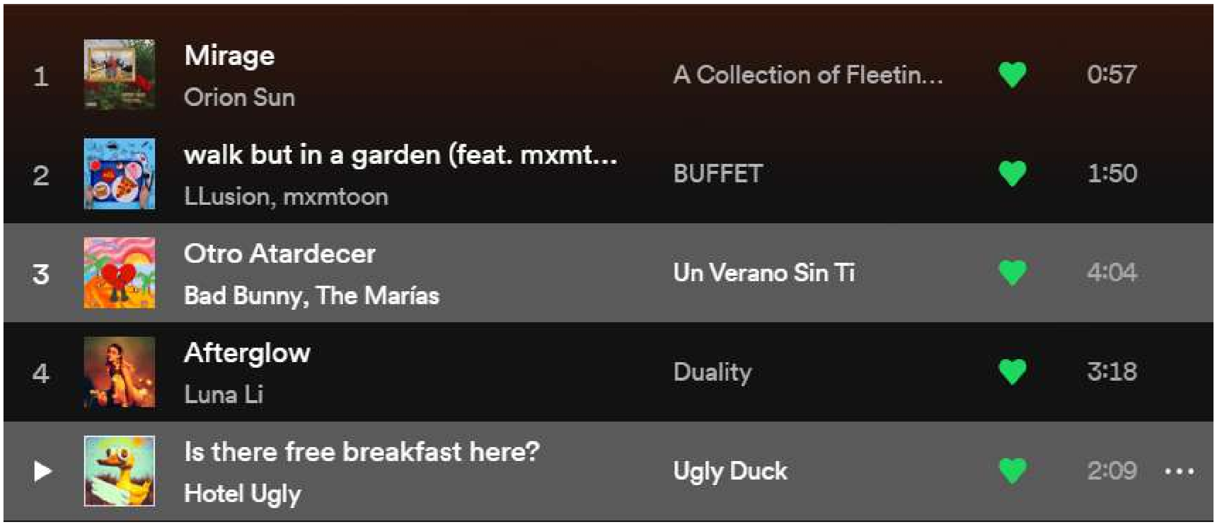
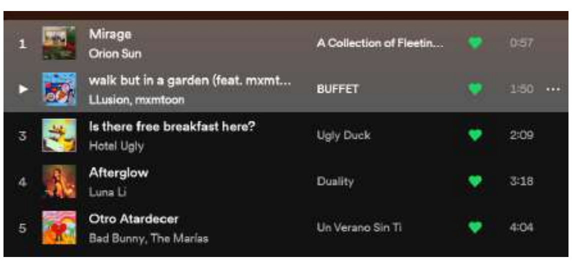
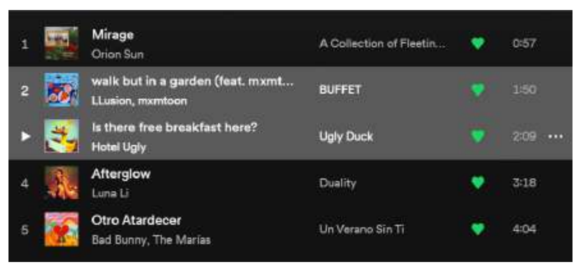


Figura 21: Ordenação da playlist

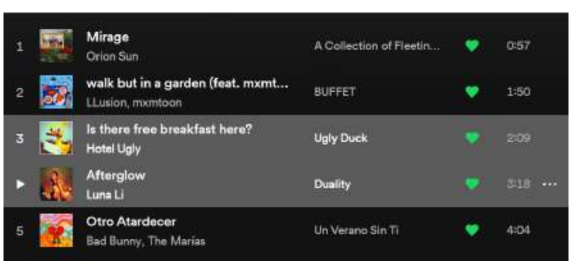
Podemos ver que a playlist ja está ordenada agora, porém o codigo continua a ser executado até o gap ser menor que 1. As proximas verificações podem ser verificadas no conjunto da figura 22.



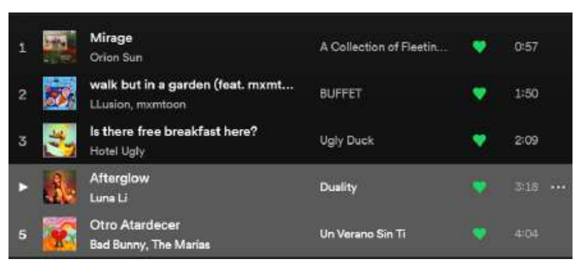
(a) Comparação do primeiro e segundo elemento



(b) Comparação do segundo e terceiro elemento



(c) Comparação do terceiro e quarto elemento



(d) Comparação do quarto e quinto elemento

Figura 22: Ordenação dos elementos

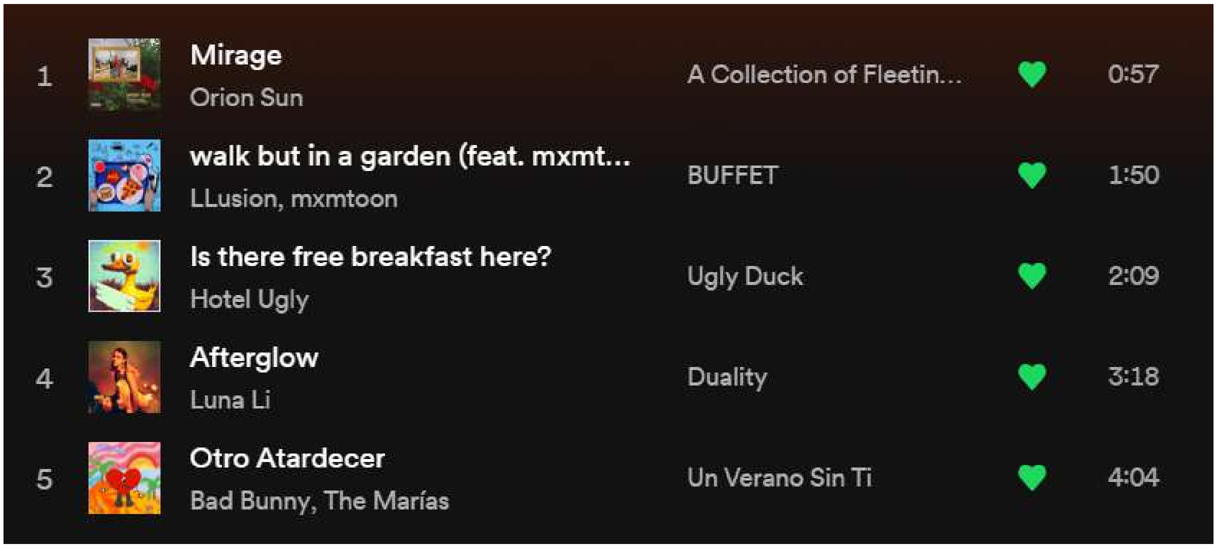


Figura 23: Playlist ordenada

3.5 *Merge Sort*

O *merge sort* é um algoritmo de ordenação um pouco diferente dos demais, ele utiliza o metodo da divisão e conquista, que consiste em dividir um problema em parcelas cada vez menores e então resolver cada uma delas, para enfim as juntar como solução do problema inicial. O merge sort faz uso deste método utilizando recursividade para dividir o problema até sua menor parcela possivel, para então juntar cada parte enquanto o resolve. O algoritmo pode ser visualizado na figura 24.

Figura 24: Código do *merge sort* em C++

Assim que o algoritmo é chamado, ele verifica se o inicio do vetor é menor que o fim, e caso seja ele armazena na variavel q o indice que representa a metade deste vetor. Em seguida a função é chamada de forma recursiva do inicio ao meio e outra vez de q+1 até o fim, criando a partir do vetor inicial dois outros, isso se repete até que restem apenas vetores com 1 elemento cada. Apos isso, é executado o merge, que ordena enquanto junta os vetores, comparando se um elemento é maior que o outro e os juntando de forma ordenada. O codigo do merge está representado na figura 25.

```cpp
void merge(int arr[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int *L = new int[n1], *M = new int[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];
    int i, j, k;
    i = 0;
    j = 0;
    k = p;
    while (i < n1 && j < n2) {
        if (L[i] < M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

Figura 25: Código do merge em C++

O algoritmo pode ser melhor representado através do exemplo das figuras 26 e 27, onde temos um vetor desordenado que devemos ordenar por meio do *merge sort*.

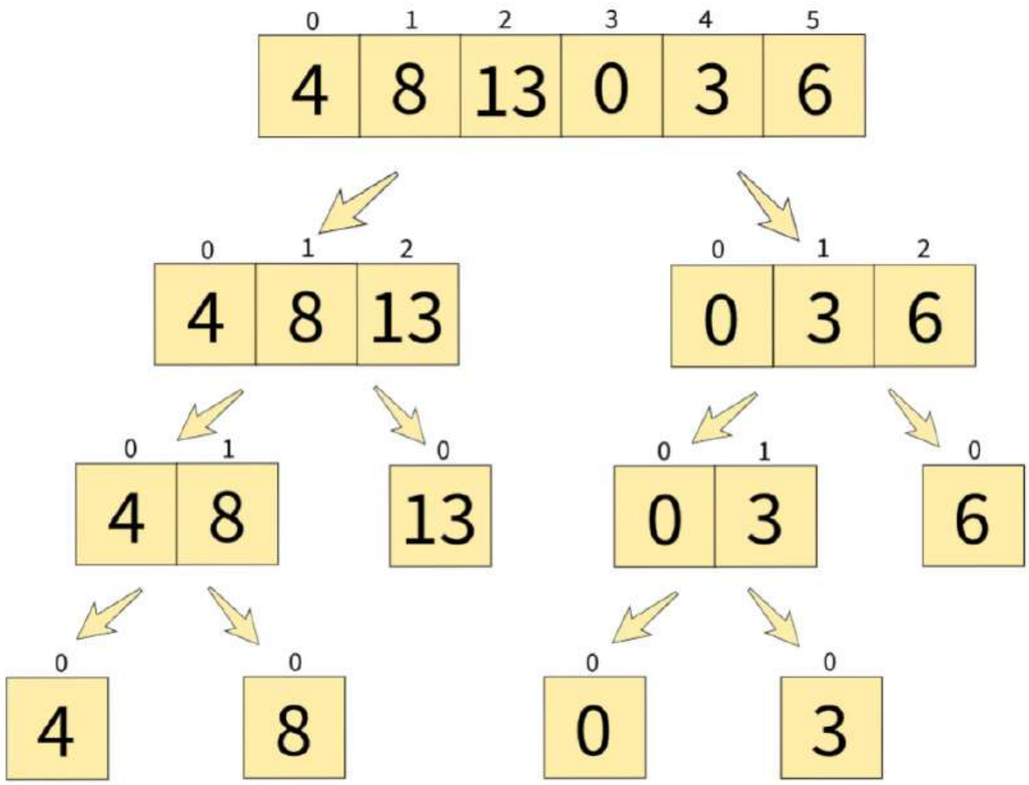


Figura 26: Exemplo de aplicação do *merge sort*

Neste exemplo, ao aplicarmos o *merge sort* dividimos o vetor em 2, como o inicio dos novos vetores e diferente do fim realizamos o mesmo passo de forma recursiva, separando-os novamente, resultando em um vetor de 2 elementos e um com apenas 1. Este vetor que possui apenas um elemento possui inicio e fim coincidentes, logo não tem necessidade de tentar dividi-lo novamente, mas, para o que possui 2 elementos esse passo é realizado novamente. Agora que não precisamos mais separar os vetores temos que junta-los por meio do merge, como demonstrado na figura 27.

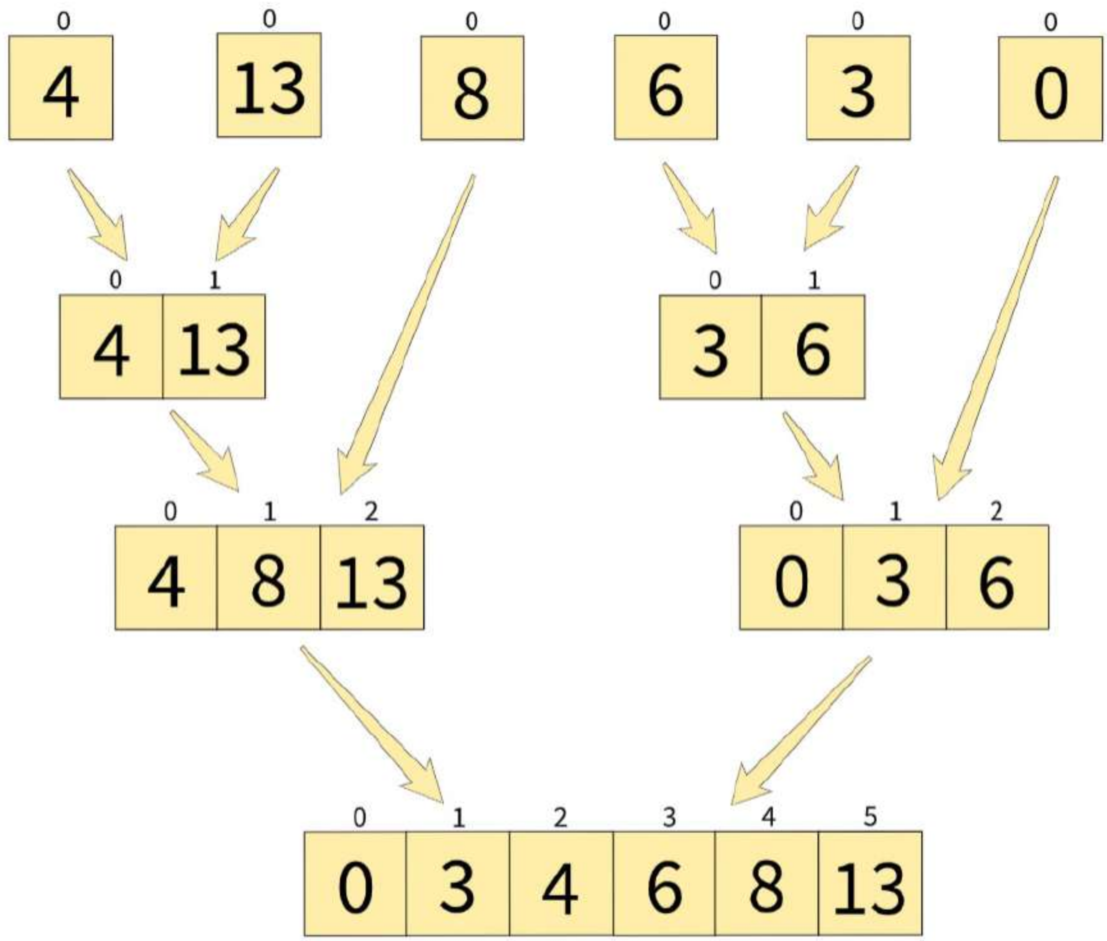


Figura 27: Exemplo de aplicação do *merge sort*

O merge faz a junção dos vetores ao mesmo tempo que ordena, comparando os elementos de cada vetor para vetificar qual é menor e colocar na posição adequada.

3.6 *Quick Sort*

O *quick sort* é um algoritmo de ordenação que, apesar de possuir um tempo de execução lento no pior caso é frequentemente a melhor opção para ordenação, devido a sua eficiencia na média do tempo esperado. O codigo do algoritmo pode ser visualizado na figura 28..

```
void quick_sort(int vetor[], int p, int r){
    if(p < r){
        int q = partition(vetor, p, r);
        quick_sort(vetor, p, q-1);
        quick_sort(vetor, q+1, r);
    }
}
```

Figura 28: Código do *quick sort* em C++

Assim como o *merge sort* esse algoritmo utiliza o método da divisão e conquista, porém, para o *quick sort* não é necessario realizar a etapa de combinação, já que a cada divisão os elementos serão colocados em sua posição ordenada. O algoritmo incialmente verifica se o indice de inicio não é o mesmo que o do final, após isso é chamada a função particiona que ordena a partir de um pivô os elementos dentro do intervalo de inicio p e fim r, colocando os números menores que o pivô atrás e maiores na frente, de forma a deixar o mesmo ordenado. Assim que o particiona termina sua execução, o indice do pivô é atribuido a uma variavel q, que é usada como base para chamar de forma recursiva o *quick sort* para o intervalo de numeros menores que o pivô e também para os maiores. O codigo do *particiona* está sendo representado na figura 29.

```
int partition(int vetor[], int p, int r){
    int q = vetor[p];
    int i = p-1;
    int j = r+1;

    while(true){
        do{
            j--;
        }while(vetor[j] > q);
        do{
            i++;
        }while(vetor[i] < q);
        if(i < j){
            troca(vetor, i, j);
        }else{
            return j;
        }
    }
}
```

Figura 29: Código do *particiona* em C++

No exemplo da figura **??** podemos ver esse algoritmo em ação, onde inicialmente temos o pivô, representado pelo q, o inicio do vetor, armazenado em i, e o fim, em j, e a partir disto incrementamos o i até encontrar um número maior que o pivô e decrementamos o j até encontrar um menor. Assim que são encontrados verificamos se i é menor que j, caso seja os dois trocam de lugar, em seguida j troca de lugar com o pivô.

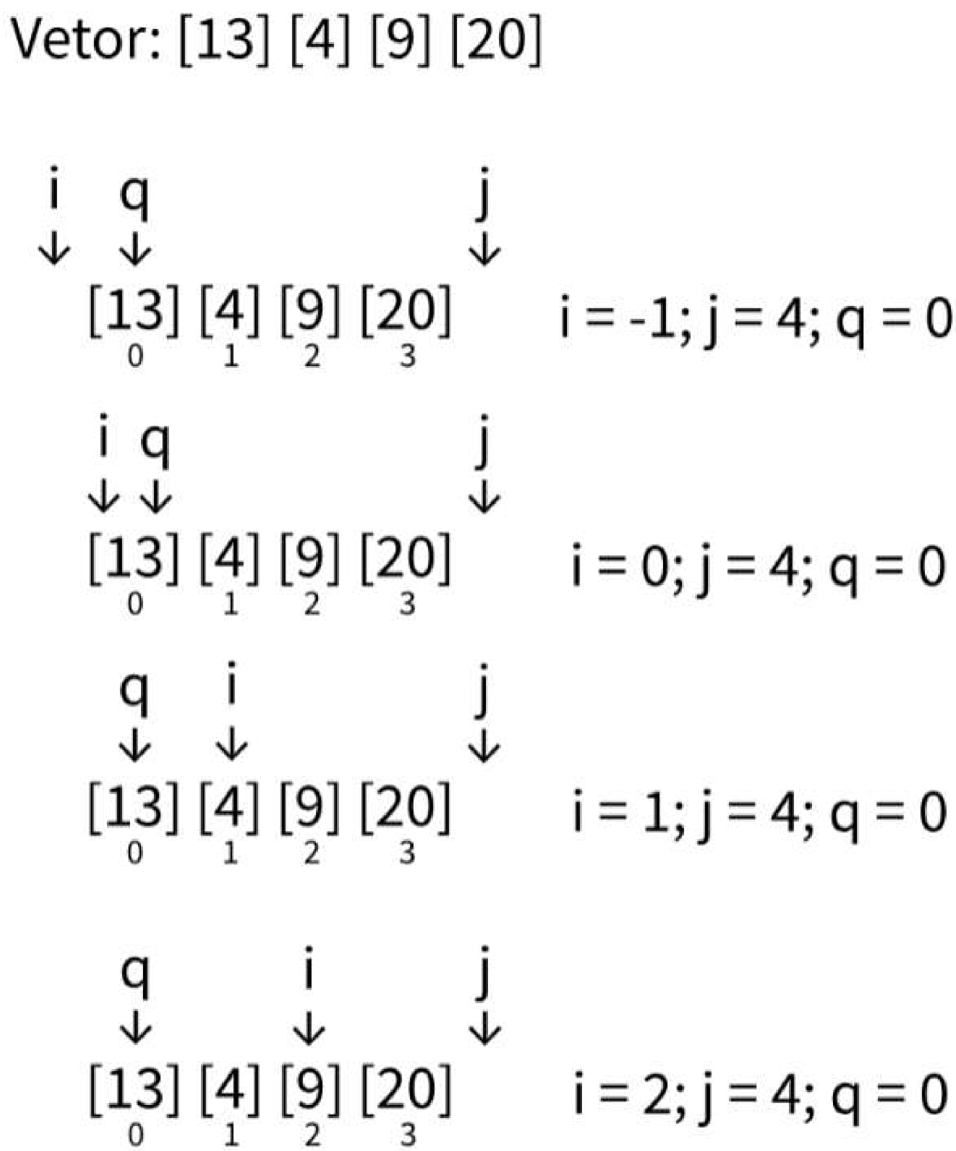


Figura 30: Execução do *quick sort*

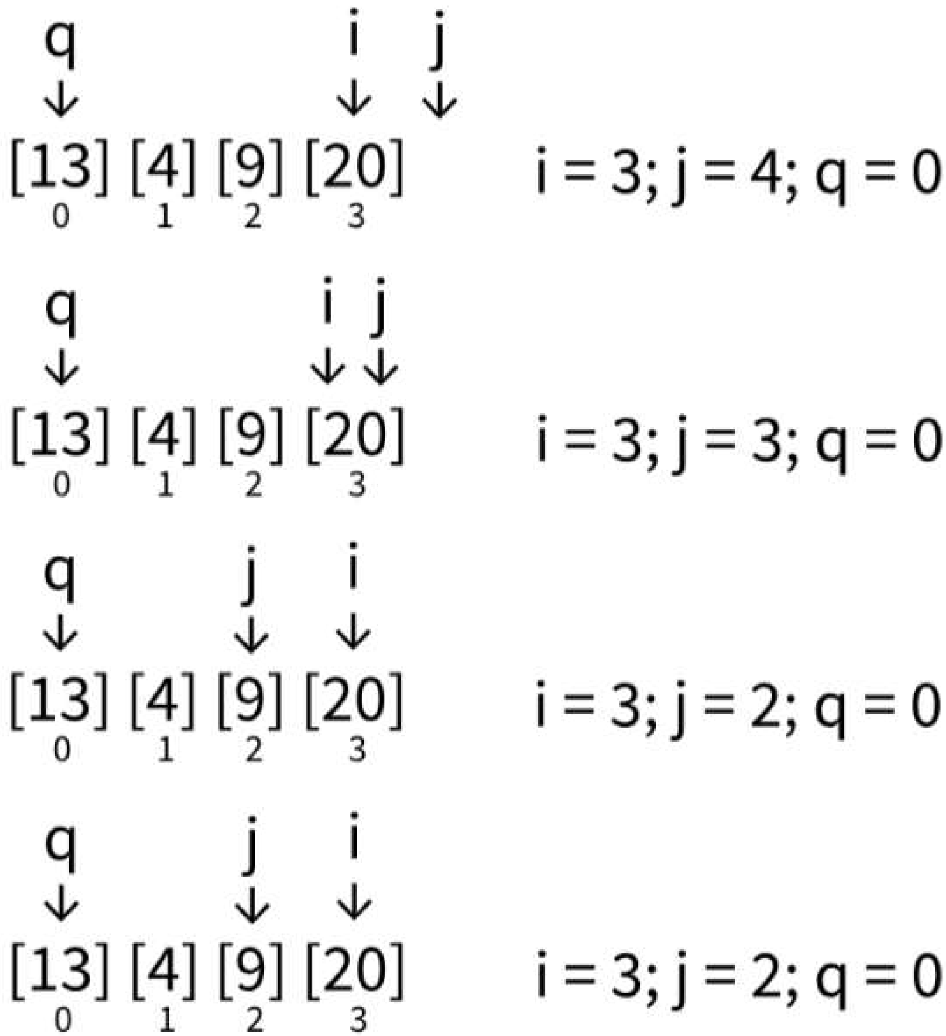


Figura 31: Execução do *quick sort*

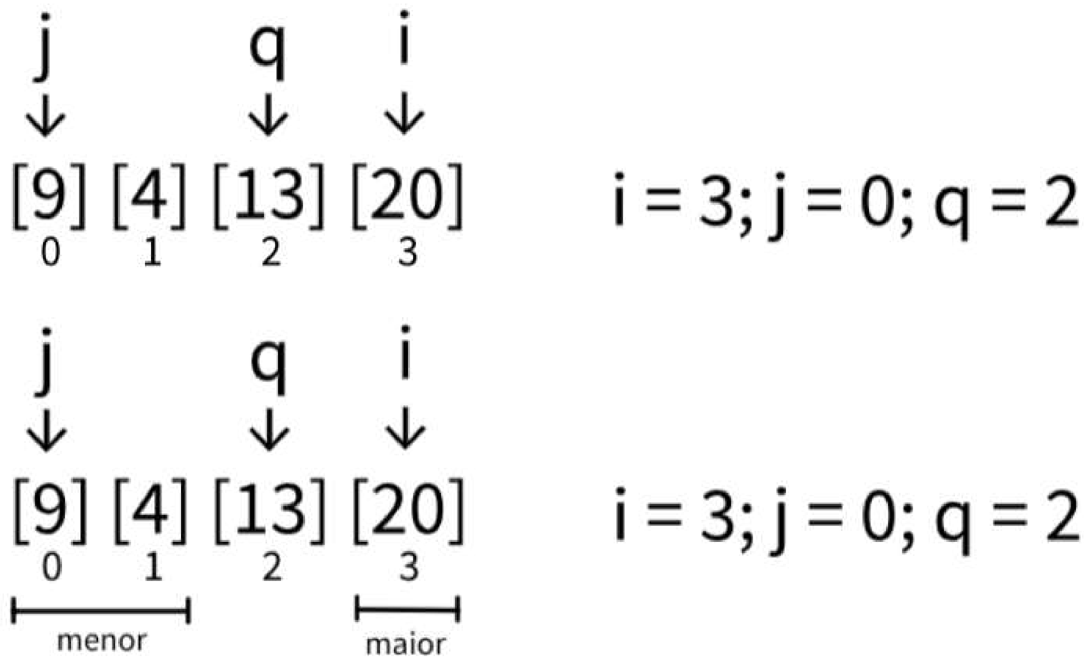


Figura 32: Execução do *quick sort*

Quando a execução do particiona acaba o pivõ se encontra ordenado, significando que todos os números menores que ele estarão à esquerda, enquanto os maiores ficarão na direita. Em seguida executamos o quick sort para mais duas vezes, uma do inicio do vetor até q-1 e de q+1 até o final dele, mas, como q+1 é igual ao final do vetor não sera executado.

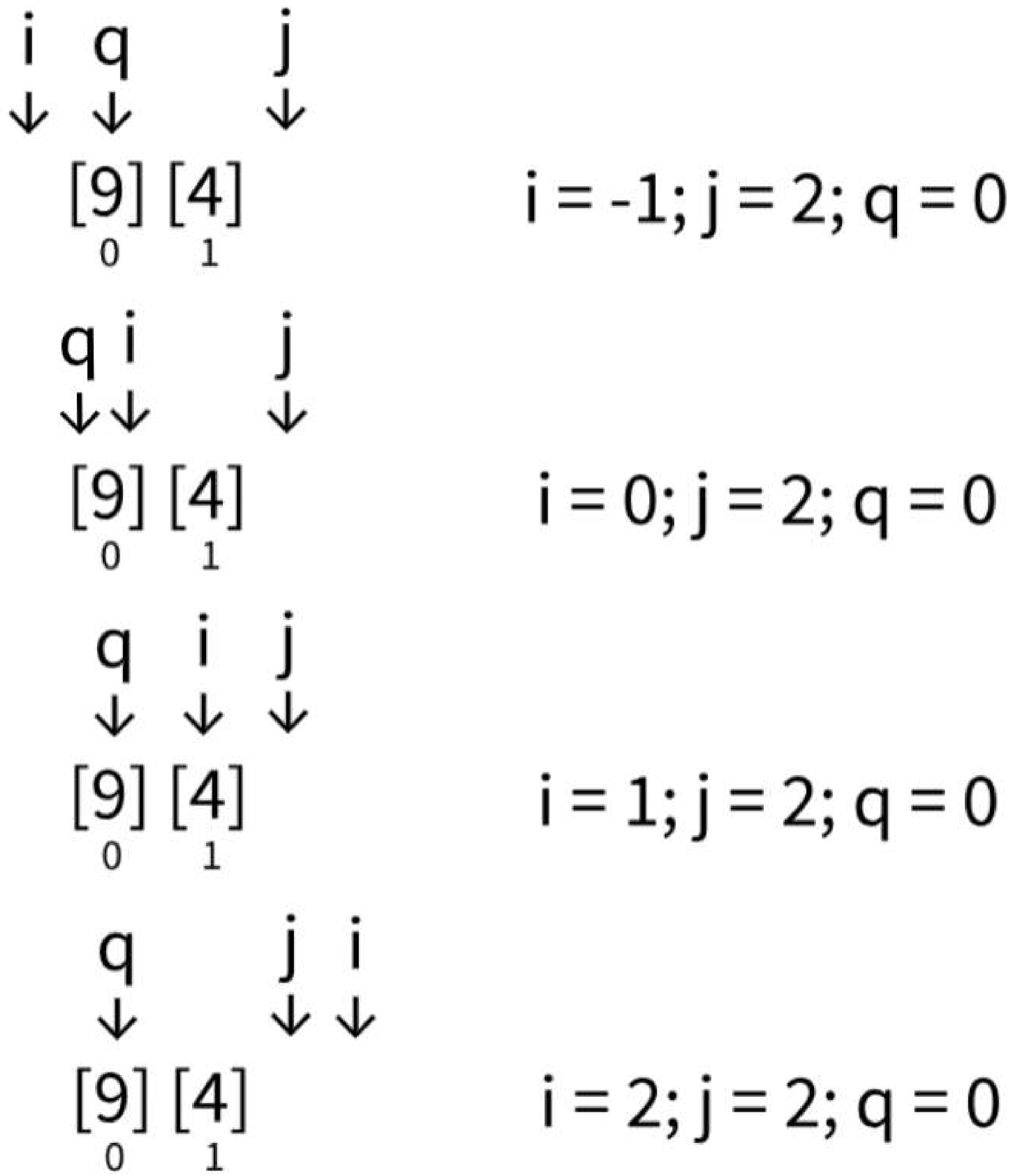


Figura 33: Execução do *quick sort*

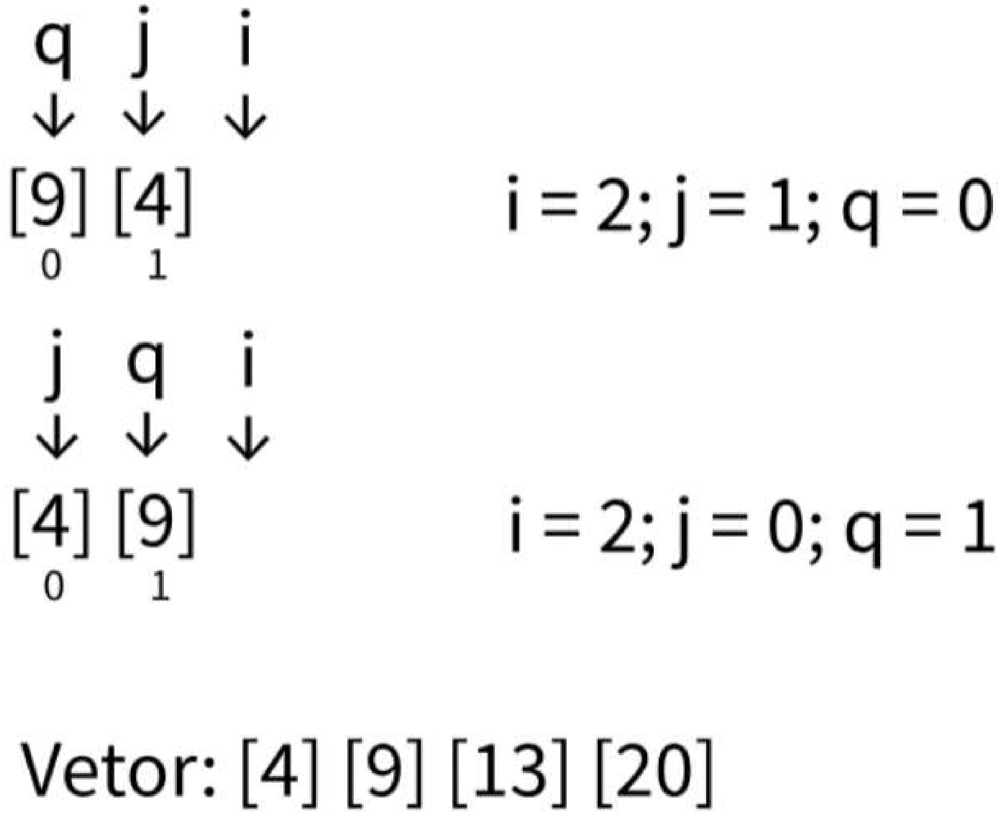


Figura 34: Execução do *quick sort*

3.6.1 *Media Como Pivô*

Apesar de que no exemplo anterior o pivô sempre era o primeiro elemento do intervalo a ser ordenado, também existem outras opções. Uma delas é utilizar a media dos indices para colocar o pivô no meio do intervalo. Um algoritmo com essa implementação pode ser verificado nas figuras 35 e 36.

```c
void middle_quicksort(int vetor[], int p, int r){
    if(p<r){
        int q = middle_partition(vetor, p, r);
        middle_quicksort(vetor, p, q-1);
        middle_quicksort(vetor, q+1, r);
    }
}
```

Figura 35: Algoritmo *quick sort* usando a media como pivô

```c
int middle_partition(int vetor[], int p, int r){
    int i;

    if(p == 0){
        i = (1+r)/2;
    }else{
        i = (p+r)/2;
    }

    troca(vetor, p, i);
    return partition(vetor, p, r);
}
```

Figura 36: Algoritmo *quick sort* usando a media como pivô

Utilizar a media como pivô pode ser benefico pois pode diminuir o consumo de memoria do algoritmo.

3.6.2 *Mediana Como Pivô*

Outra opção para encontrar um pivô é utilizar a mediana, neste caso é realizada uma comparação entre o primeiro elemento, o ultimo e o que é o resulltado da media entre estes dois. Esta comparação verifica dentre os três o maior e o menor número, estes são descartados e é utilizado como pivô aquele que restou. A implementação está nas figuras 37 e 38.

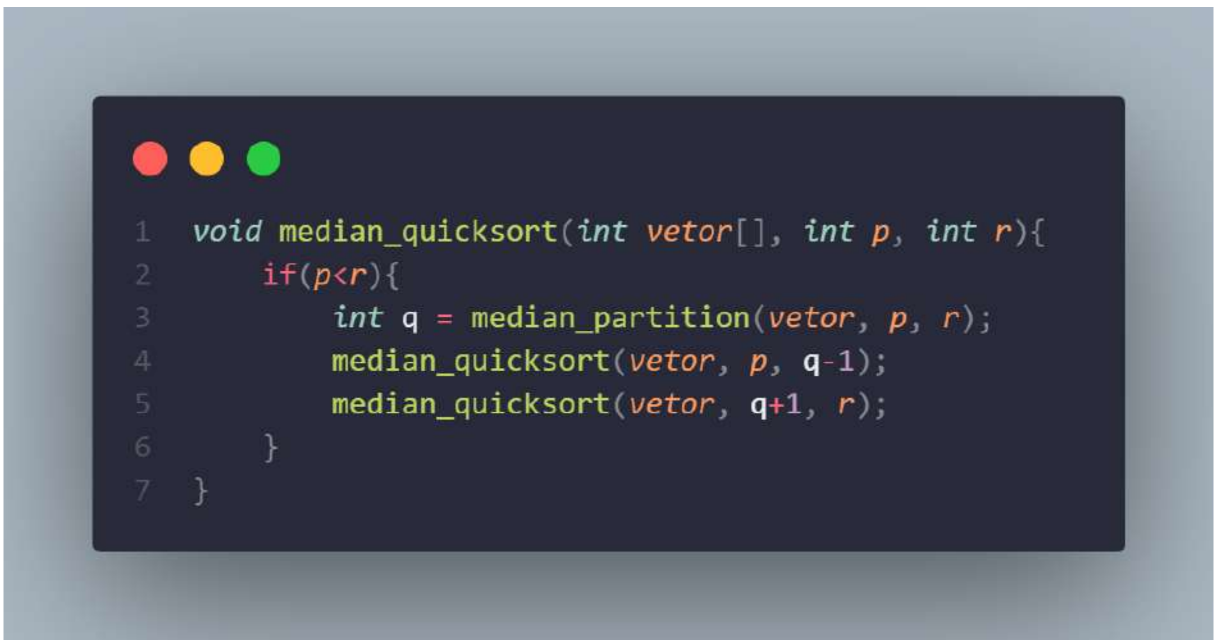


Figura 37: Algoritmo *quick sort* usando a mediana como pivô

```c
int median_partition(int vetor[], int p, int r){
    int i, med;

    if(p == 0){
        i = (1+r)/2;
    }else{
        i = (p+r)/2;
    }

    if(vetor[p] < vetor[i]){
        if(vetor[i] < vetor[r]){
            med = i;
        }else if(vetor[p] < vetor[r]){
            med = r;
        }else{
            med = p;
        }
    }else if(vetor[r] < vetor[i]){
        med = i;
    }else if(vetor[r] < vetor[p]){
        med = r;
    }else{
        med = p;
    }

    troca(vetor, p, med);
    return partition(vetor, p, r);
}
```

Figura 38: Algoritmo *quick sort* usando a mediana como pivô

3.6.3 *Pivô Aleatorio*

Uma opção que também se mostra interessante é usar um indice aleatorio dentre os presentes no intervalo para ser usado como pivô, como observado nas figuras abaixo.

```
void random_quicksort(int vetor[], int p, int r){
    if(p<r){
        int q = random_partition(vetor, p, r);
        random_quicksort(vetor, p, q-1);
        random_quicksort(vetor, q+1, r);
    }
}
```

Figura 39: Algoritmo *quick sort* usando um pivô aleatorio

```
int random_partition(int vetor[], int p, int r){
    int i = random(p, r);
    troca(vetor, r, i);
    return partition(vetor, p, r);
}
```

Figura 40: Algoritmo *quick sort* usando um pivô aleatorio

3.7 *Heap Sort*

O *heap sort* é um algoritmo de ordenação que utiliza uma estrutura de dados chamada heap, sendo esssa baseada em um modelo de representação de array em árvore binária. Neste modelo cada nó da árvore corresponde a um elemento do arranjo, sendo que todos os níveis estão preenchidos, exceto o ultimo. Esta árvore pode ser ordenada de forma que a raiz sejá o menor ou o maior elemento, para este algoritmo nossa raiz será o menor elemento do array.

```cpp
void min_heapify(vector<int> &vetor, int tamanho, int i){
    int l = 2*i+1, r = 2*i+2;
    int menor = i;

    if(l <= tamanho && vetor[l] < vetor[menor]){
        menor = l;
    }

    if(r <= tamanho && vetor[r] < vetor[menor]){
        menor = r;
    }

    if(menor != i){
        swap(vetor[i], vetor[menor]);
        min_heapify(vetor, tamanho, menor);
    }
}

void build_min_heap(vector<int> &vetor) {
    int tamanho = vetor.size() - 1;
    for (int i = (vetor.size() - 1) / 2; i >= 0; i--) {
        min_heapify(vetor, tamanho, i);
    }
}
```

Figura 41: Código do *min heapify* e do *build min heap* em C++

Para construirmos então uma árvore binária a partir de um array utilizaremos as duas funções da figura 41, o *min heapify*, que ordena um elemento seguindo a regra de heap minimo, onde o nó referido tem que ser menor que seu filho da esquerda e da direita e o *build min* *heap*, que executa o *min heapify* para todos os elementos, do último ao primeiro. Isso resulta em uma arvore em que todos os nós pai são menores que seus filhos.

```cpp
void heapSort(vector<int> &vetor) {
    build_min_heap(vetor);
    int tamanho = vetor.size() - 1;
    for (int i = vetor.size() - 1; i > 0; i--) {
        swap(vetor[0], vetor[i]);
        tamanho--;
        min_heapify(vetor, tamanho, 0);
    }
}
```

Figura 42: Código do *heap sort* em C++

Para ordenar os elementos de um vetor usando desse modelo nós usamos a função *heap sort* representada na figura 42, onde, após o heap mínimo ser construido o primeiro elemento do vetor, que no caso é o menor, troca de lugar com o último e logo em seguida o *min heapify* é executado para ordenar o novo nó raiz, mas sem considerar o elemento já ordenado. Isso se repete em loop n-1 vezes, onde n é o tamanho do vetor, resultando em um vetor ordenado de forma decrescente ao final da execução.

Partindo para um exemplo visual do funcionamento do *heap sort* podemos ver a execução do *build min heap* na figura 43, onde o vetor é ordenado seguindo as regras da árvore binária mínima.

Agora que o heap mínimo foi construido podemos começar a ordenação conforme as figuras 44, 45, 46 e 47, onde vamos "reservar"o menor elemento de cada execução do *min heapify*.

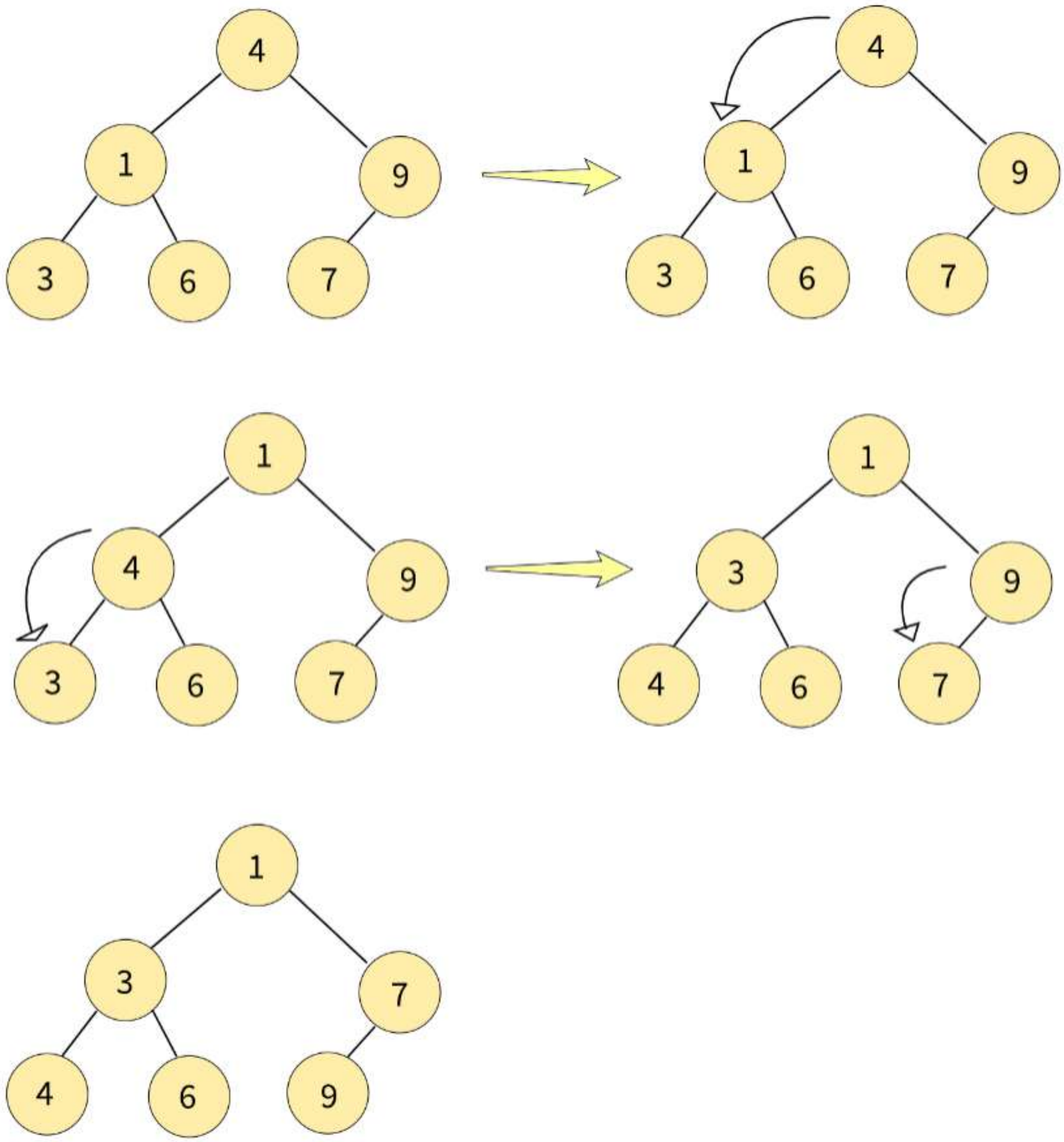


Figura 43: Execução do *build min heap*

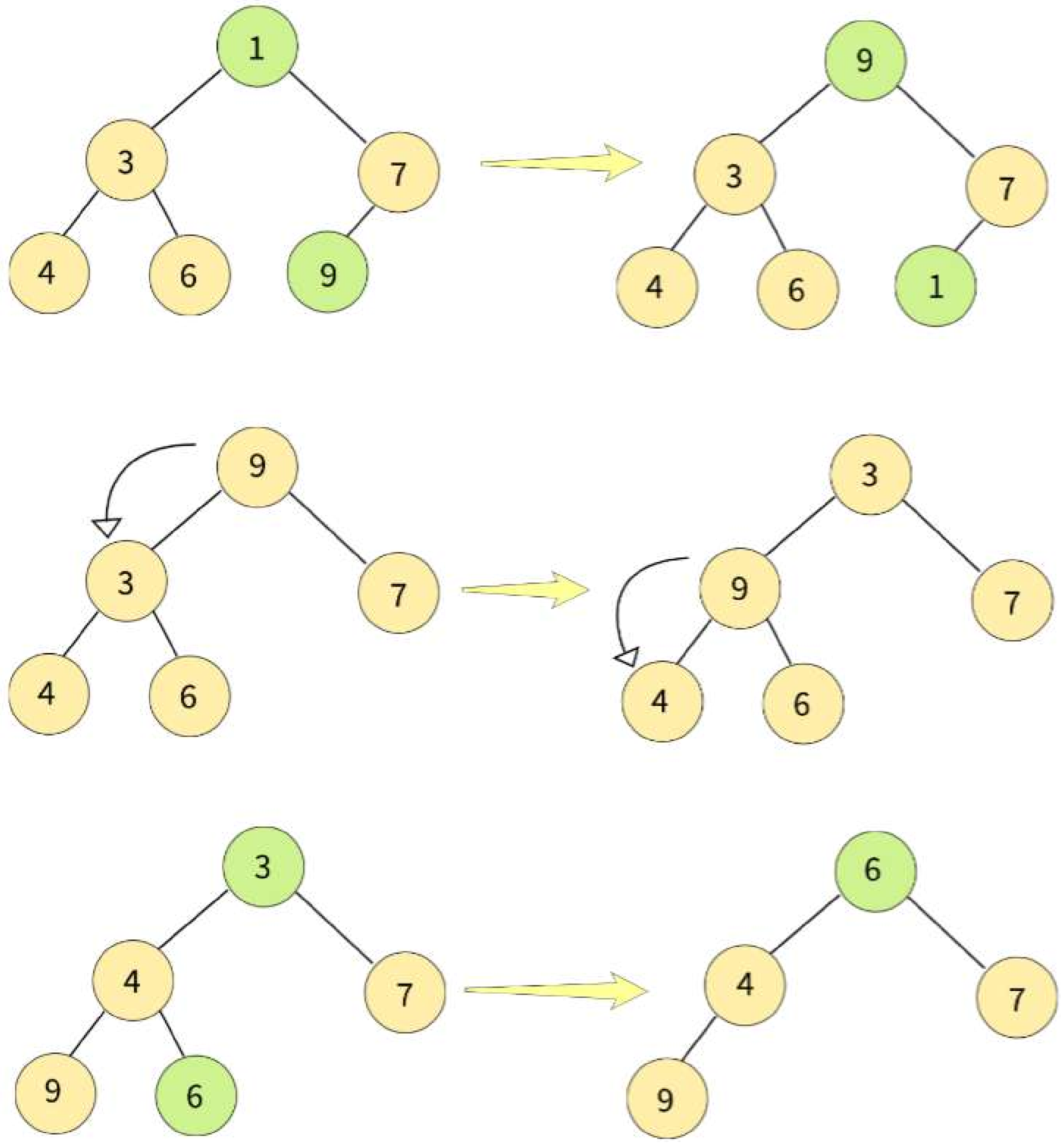


Figura 44: Execução do *heap sort*

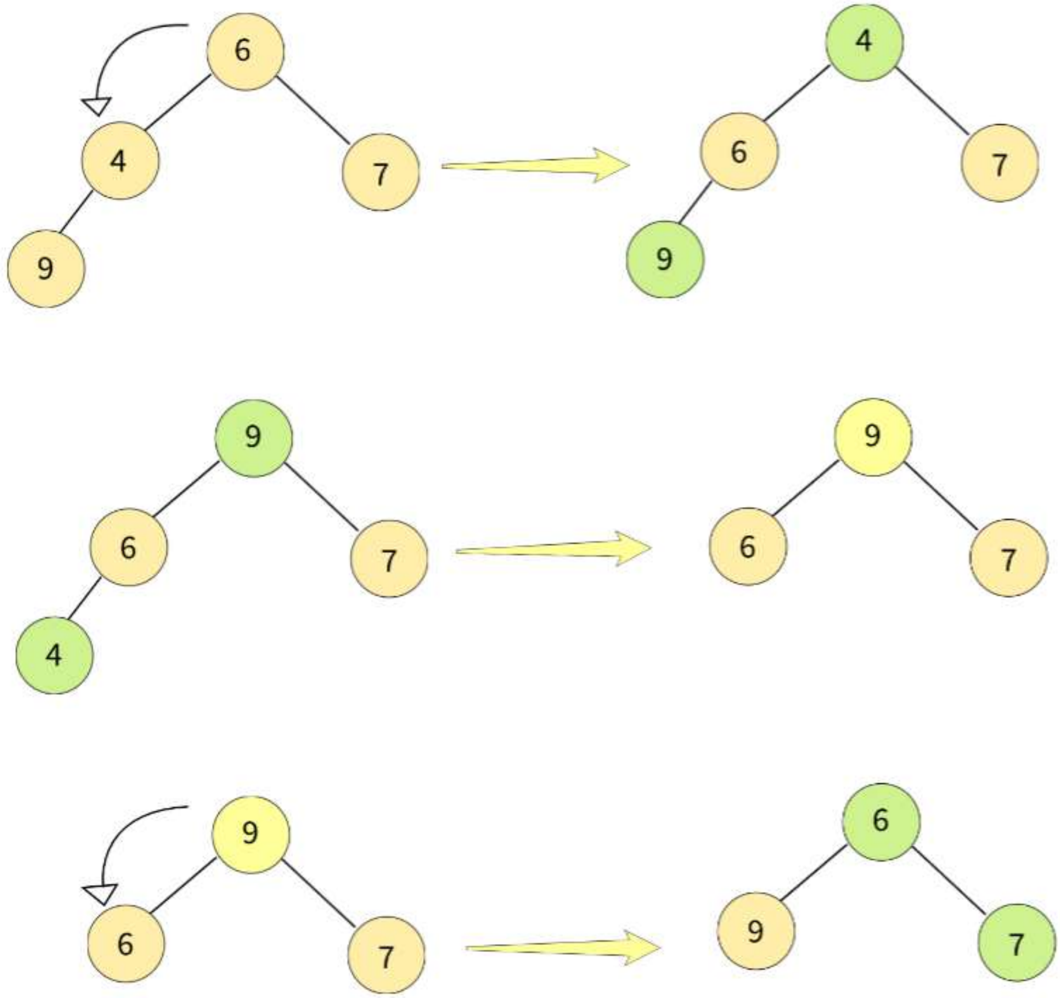


Figura 45: Execução do *heap sort*

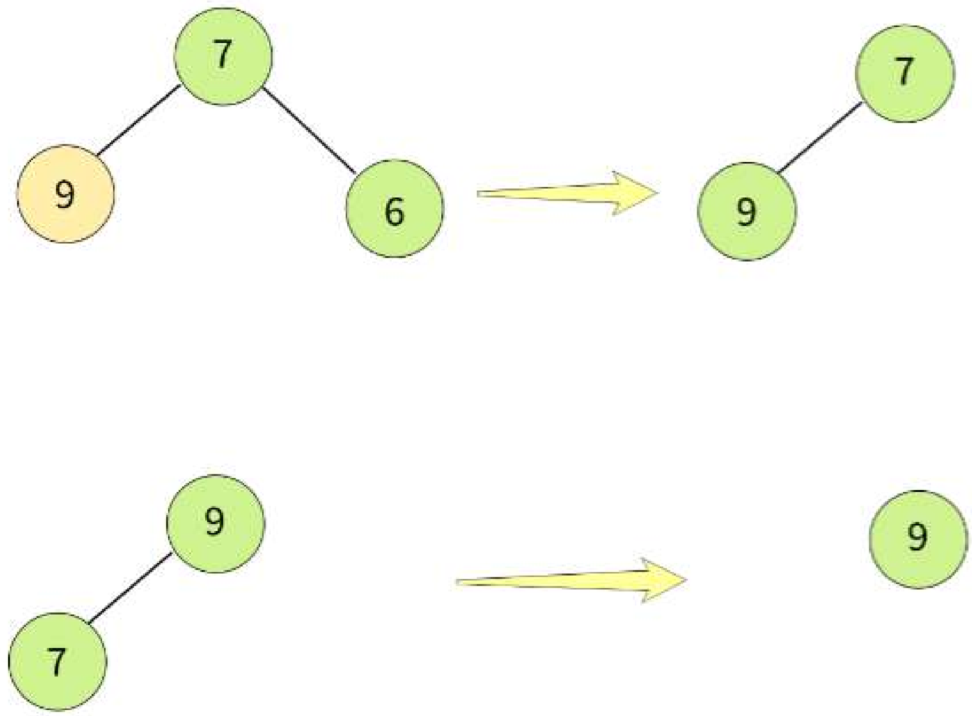


Figura 46: Execução do *heap sort*

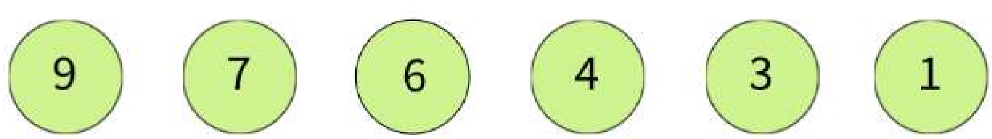


Figura 47: Vetor ordenado

4 ANÁLISE DE COMPLEXIDADE

4.1 *Insertion Sort*

Para realizar a análise de complexidade do *insertion sort* primeiro é necessário definir quanto tempo leva para executar o algorítimo uma vez, mas como não é possível saber exatamente quanto tempo diferentes maquinas vão levar para processar cada linha temos que chegar em uma equação geral, portanto, o valor de cada linha sera igual a uma variável.

```
for(j = 1; j < tamanho; j++){
        chave = vetor[j];
        i = j - 1;
        while(i >= 0 && vetor[i] > chave){
            vetor[i+1] = vetor[i];
            i--;
        }
        vetor[i+1] = chave;
    }
```

Figura 48: Algoritmo *insertion sort* em C++

Temos 9 linhas de código no total, cada uma delas sera associada a uma variável c diferente que representa o tempo de execução da mesma:

Linha	Codigo	Custo	Número de Execuções
1	For(j = 1; i < tamanho; j++){	C1	N
2	chave = vetor[j]	C2	N-1
3	i = j - 1	C3	N-1
4	while(i >= 0 && vetor[i] > chave){	C4	$\sum_{i=2}^{N} Tj$
5	vetor[i+1] = vetor[i]	C5	$\sum_{i=2}^{N} Tj - 1$
6	$i--$	C6	$\sum_{i=2}^{N} Tj - 1$
7	}	C7	$\sum_{i=2}^{N} Tj - 1$
8	vetor[i+1] = chave	C8	N-1
9	}	C9	N-1

Linha 1 - Como é a primeira linha do programa não se sabe ao certo quantas vezes ela será executada já que isso depende do tamanho do vetor inserido, portanto definiremos essa variável como N, logo C1 sera executada N vezes.

Linha 2 - C2 sera executada uma vez a menos que C1, pois é realizada a verificação para sair do for uma vez a mais antes do programa encerrar, então C2 sera executada N-1 vezes.

Linha 3 - C3 segue a mesma logica de C2, ela sera executada n-1 vezes.

Linha 4 - Não é possível saber quantas vezes esse trecho do código vai ser executado, pois além da variável N ele também depende de como os elementos estão ordenados dentro do vetor e como não se pode representar isso com apenas outra variável vamos utilizar um somatório de Tj, que é o número de vezes que o while foi executado para um valor de j, variando de 1 até N. Então C4 sera executada $\sum_{i=2}^{N} Tj$ vezes.

Linha 5 - C5 segue a mesma logica de C4, porem ela sera executada uma vez a menos para cada vez que o programa entrar no while, logo ela sera executada $\sum_{i=2}^{N} Tj - 1$ vezes .

Linha 6 - C6 seguem a mesma logica de C5, ela sera executada $\sum_{i=2}^{N} Tj - 1$ vezes.

Linha 7 - Essa é executada porem não é calculada já que é apenas o fechamento do while, portanto seu tempo de execução e ínfimo.

Linha 8 - Como C8 já esta fora do while essa linha será executada N-1 vezes.

Linha 9 - Essa é executada porem não é calculada já que é apenas o fechamento do for, portanto seu tempo de execução e ínfimo.

Agora que é possível calcular o tempo gasto em cada linha é necessário apenas considerar os possíveis casos de entrada de dados para avaliar cada um de forma independente. Para calcular o tempo que sera gasto basta somar o produto do tempo das linha pelo número de execuções em uma função tempo t(n) e considerar as entradas de cada caso. os casos são:

Melhor caso - caso em que o vetor já esta ordenado, logo nenhuma mudança é necessária, resultando na execução mais rápida possível. Neste caso as linhas 5 e 6 são desconsideradas, pois não existe a necessidade de ordenar um vetor já ordenado, a 4 sera considerada já que ainda é feita a verificação, mas ela será executada N-1 vezes.

t(n) = $C1 * N + C2 * (N-1) + C3 * (N-1) + C4 * (N-1) + C8 * (N-1)$

t(n) = $C1 * N + C2 * N + C3 * N + C4 * N + C8 * N - C2 - C3 - C4 - C8$

t(n) = $N * (C1 + C2 + C3 + C4 + C8) + (-C2 - C3 - C3 - C8)$

Com isso podemos perceber que o melhor caso é representado por uma função linear $A * N + B$, onde A é igual aos números que multiplicam N e B os números que estão subtraindo.

Pior caso - Caso em que os números do vetor estão ordenados em ordem crescente, sendo necessário que o programa ordene todos os números, resultando no maior gasto de tempo possível. Como é necessário executar o while o máximo de vezes possíveis em cada execução vamos considerar que Tj é igual a j.

$\sum_{i=2}^{N} Tj = \sum_{i=2}^{N} j$ = (2, 3, 4, ..., N)

S = $\frac{N*(a1+an)}{2}$

S = $\frac{N^2+N-2)}{2}$

$\sum_{i=2}^{N} Tj - 1 = \sum_{i=2}^{N} j + \sum_{i=2}^{N} 1$

$\frac{N^2+N-2)}{2} - N - 1 = \frac{N^2-N}{2}$

t(n) = $C1 * N + C2 * (N-1) + C3 * (N-1) + C4 * \frac{N^2+N-2)}{2} + C6 * \frac{N^2-N)}{2} + C7 * \frac{N^2-N)}{2} + C8 * (N-1)$

t(n) = $N^2 * (\frac{C4*N^2)}{2} + \frac{C5*N^2)}{2} + \frac{C6*N^2)}{2}) + N * (C1 + C2 + C3 + \frac{C4*N)}{2} - \frac{C6*N)}{2} - \frac{C7*N)}{2} + C8) + (-C2 - C3 - C4 - C8)$

O pior caso é representado por uma função quadrática $A * N^2$ + $B * N$ + C, onde A são os números que multiplicam N^2, B são os números que multiplicam N e C os números que estão subtraindo.

Caso médio - Caso comum, onde sera necessário ordenar a maior parte dos números, tendo uma execução não tão rapida mas também não tão lenta. É o resultado da metade das execuções máximas do while.

$\frac{\frac{N^2+N-2}{2}}{2} = \frac{N^2+N-2}{4}$

$\frac{\frac{N^2-N}{2}}{2} = \frac{N^2-N}{4}$

t(n) $= C1 * N + C2 * (N-1) + C3 * (N-1) + C4 * \frac{N^2+N-2)}{4} + C6 * \frac{N^2-N)}{4} + C7 * \frac{N^2-N)}{4} + C8 * (N-1)$

t(n) $= N^2 * (\frac{C4*N^2)}{4} + \frac{C5*N^2)}{4} + \frac{C6*N^2)}{4}) + N * (C1 + C2 + C3 + \frac{C4*N)}{4} - \frac{C6*N)}{4} - \frac{C7*N)}{4} + C8) + (-C2 - C3 - C4 - C8)$

Assim como no pior caso o caso médio é representado por uma função quadrática $A * N^2$ + $B * N$ + C, onde A são os números que multiplicam N^2, B são os números que multiplicam N e C os números que estão subtraindo.

Com esses cálculos podemos definir o "teto"e o "piso"do algorítimo, ou seja, o tempo máximo e mínimo de execução. Isso traz a certeza de que para qualquer entrada o programa nunca vai demorar mais do que o teto e nunca menos que o piso.

4.2 *Bubble Sort*

Para a análise de complexidade do *bubble sort* será usado o mesmo método, utilizando variaveis para representar o tempo de execução de cada linha e considerar também o número de execuções em cada caso. A parte a ser considerada durante a execução do codigo será apenas a representada na figura 49.

Primeiramente é necessario definir os custos das linhas, que sera representado por uma variavel C para cada linha de codigo executavel, em seguida precisamos dos números de execução que será calculado com base no número de execuções da primeira linha como pode

Figura 49: Algoritmo *bubble sort*

ser visto na tabela a seguir.

Linha	Codigo	Custo	Número de Execuções
1	For(i = 1; i < (tamanho - 1); i++)	C1	N
2	For(j = 0; j < (tamanho - i); j++)	C2	N-1
3	If(vetor[j] < vetor[j + 1])	C3	$\sum_{i=1}^{N-2} tj$
4	chave = vetor[j]	C4	$\sum_{i=1}^{N-2} tj - 1$
5	vetor[j] = vetor[j + 1]	C5	$\sum_{i=1}^{N-2} tj - 1$
6	vetor[j + 1] = chave	C6	$\sum_{i=1}^{N-2} tj - 1$

Agora que possuimos todo o necessario basta considerar o pior, melhor e medio caso para o algoritmo e estimar o número de execuções com base nele.

Melhor caso - No melhor caso não é necessario executar as linhas 4, 5 e 6 pois o vetor ja está ordenado.

$t(n) = C1 * N + C2 * (N - 1) + C3 * \sum_{i=1}^{N-2} tj$

$\sum_{i=1}^{N-2} tj = (2, 3, 4, ..., (N-2))$

S = $\frac{(N-2)*(a1+an)}{2}$

S = $\frac{(N-2)*(2+(N-2)}{2}$

S = $\frac{(N^2+2*N-2*N+4-4}{2}$

S = $\frac{N^2}{2}$

t(n) = $C1 * N + C2 * (N-1) + C3 * \frac{N^2}{2}$

t(n) = $N^2 * (C3) + N * (C1 + C2) + (-C2)$

No melhor caso desse algoritmo temos uma função quadrática $A * N^2 + B * N + C$, onde A são os números que multiplicam N^2, B são os números que multiplicam N e C os números que estão subtraindo.

Pior caso - Neste caso o vetor esta ordenado de forma decrescente, necessitando do número maximo de execuções para colocar o mesmo em forma crescente.

t(n) = $C1 * N + C2 * (N-1) + C3 * \sum_{i=1}^{N-2} tj + C4 * \sum_{i=1}^{N-2} tj - 1 + C5 * \sum_{i=1}^{N-2} tj - 1 + C6 * \sum_{i=1}^{N-2} tj - 1$

$\sum_{i=1}^{N-2} tj - 1 = \sum_{i=1}^{N-2} tj - \sum_{i=1}^{N-2} 1$

$= \frac{N^2}{2} - (N-1) = \frac{N^2-2*N-2}{2}$

t(n) = C1*N + C2*(N-1) + $C3 * \frac{N^2}{2} + C4 * \frac{N^2-2*N-2}{2} + C5 * \frac{N^2-2*N-2}{2} + C6 * \frac{N^2-2*N-2}{2}$

t(n) = $N^2 * (\frac{C3}{2} + \frac{C4}{2} + \frac{C5}{2} + \frac{C6}{2}) + N * (C1 + C2 - C4 - C5 - C6) + (-C2 - C4 - C5 - C6)$

O pior caso é representado por uma função quadrática $A * N^2 + B * N + C$, onde A são os números que multiplicam N^2, B são os números que multiplicam N e C os números que estão subtraindo.

Medio caso - Para representar este caso basta pegar o número de execuções das linhas 4, 5 e 6 e contar com metade das execuções maximas.

t(n) = $C1 * N + C2 * (N-1) + C3 * \frac{\sum_{i=1}^{N-2} tj}{2} + C4 * \frac{\sum_{i=1}^{N-2} tj - 1}{2} + C5 * \frac{\sum_{i=1}^{N-2} tj - 1}{2} + C6 * \frac{\sum_{i=1}^{N-2} tj - 1}{2}$

t(n) = C1*N + C2*(N-1) + $C3 * \frac{N^2}{4} + C4 * \frac{N^2-4*N-4}{4} + C5 * \frac{N^2-4*N-4}{4} + C6 * \frac{N^2-4*N-4}{4}$

t(n) = $N^2 * (\frac{C3}{4} + \frac{C4}{4} + \frac{C5}{4} + \frac{C6}{4}) + N * (C1 + C2 - C4 - C5 - C6) + (-C2 - C4 - C5 - C6)$

Assim como no pior e melhor caso o caso medio é representado por uma função quadrática $A * N^2 + B * N + C$, onde A são os números que multiplicam N^2, B são os números que multiplicam N e C os números que estão subtraindo.

4.3 *Selection Sort*

Para a análise do *selection sort* inicialmente devemos pegar as partes do codigo que serão relevantes para nós, conforme a figura 50.

```
for(i = 0; i < tamanho; i++){
    aux = i;
    for(j = (i + 1); j < tamanho; j++){
        if(vetor[j] < vetor[aux]){
            aux = j;
        }
    }
    chave = vetor[i];
    vetor[i] = vetor[aux];
    vetor[aux] = chave;
}
```

Figura 50: Algoritmo *selection sort* em C++

Agora calcularemos o tempo gasto em cada linha, para isso será atribuida uma variavel C para cada uma delas, além disso também calcularemos o número de execuções de cada caso com base no número de execuções da primeira linha.

Linha	Codigo	Custo	Número de Execuções
1	For(i = 0; i < tamanho; i++)	C1	N
2	aux = i	C2	N-1
3	For(j = (i + 1); j ¡ tamanho; j++)	C3	$\sum_{i=1}^{N-1} tj$
4	if(vetor[j] < vetor[aux])	C4	$\sum_{i=1}^{N-2} tj - 1$
5	aux = j	C5	$\sum_{i=1}^{N-2} tj - 1$
6	chave = vetor[i]	C6	N-1
7	vetor[i] = vetor[aux]	C7	N-1
8	vetor[aux] = chave	C8	N-1

Melhor caso - Para esse calculo devemos considerar que o vetor já está ordenado, e consequentemente a linha 5 não sera executada.

t(n) $= C1 * N + C2 * (N-1) + C3 * \sum_{i=1}^{N-1} tj + C4 * \sum_{i=1}^{N-2} tj + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$\sum_{i=1}^{N-1} tj = (1, 2, 3, ..., (N-1))$

$S = \frac{N*(a1+an)}{2} = \frac{(N-1)*(1+(N-1))}{2}$

$S = \frac{N^2+N-2*N+1-1}{2} = \frac{N^2-N}{2}$

$\sum_{i=1}^{N-2} tj = (1, 2, 3, ..., (N-2))$

$S = \frac{N*(a1+an)}{2} = \frac{(N-2)*(1+(N-2))}{2}$

$S = \frac{N^2+N-4*N+4-2}{2} = \frac{N^2-3*N+2}{2}$

$\sum_{i=1}^{N-2} tj - 1 = \sum_{i=1}^{N-2} tj - \sum_{i=1}^{N-2} 1$

$\frac{N^2-3*N+2}{2} - (N-2) = \frac{N^2-3*N+2-2*N+4}{2} = \frac{N^2-5*N+6}{2}$

$t(n) = C1 * N + C2 * (N-1) + C3 * \frac{N^2-N}{2} + C4 * \frac{N^2-3*N+2}{2} + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$t(n) = N^2 * (\frac{C3}{2} + \frac{C4}{2}) + N * (C1 + C2 - \frac{C3}{2} - \frac{3*C4}{2} + C6 + C7 + C8) + (-C2 + \frac{2*C4}{2} - C6 - C7 - C8)$

A equação que representa o melhor caso é quadratica $A * N^2 + B * N + C$.

Pior caso - Neste caso o vetor esta na ordem decrescente, necessitando do máximo de execuções do algoritmo.

t(n) $= C1 * N + C2 * (N-1) + C3 * \sum_{i=1}^{N-1} tj + C4 * \sum_{i=1}^{N-2} tj + C5 * \sum_{i=1}^{N-2} tj + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$t(n) = C1 * N + C2 * (N-1) + C3 * \frac{N^2-N}{2} + C4 * \frac{N^2-3*N+2}{2} + C5 * \frac{N^2-3*N+2}{2} + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$t(n) = N^2 * (\frac{C3}{2} + \frac{C4}{2} + \frac{C5}{2}) + N * (C1 + C2 - \frac{C3}{2} - \frac{3*C4}{2} - \frac{3*C5}{2} + C6 + C7 + C8) + (-C2 + \frac{2*C4}{2} + \frac{2*C5}{2} - C6 - C7 - C8)$

Assim como no melhor caso, a função da complexidade do pior caso deste algoritmo é uma função quadratica $A * N^2 + B * N + C$.

Medio caso - Para calcular este caso basta considerarmos que para ordenar o vetor serão necessarias metade das execuções máximas.

t(n) $= C1 * N + C2 * (N-1) + C3 * \sum_{i=1}^{N-1} tj + C4 * \sum_{i=1}^{N-2} tj + C5 * \frac{\sum_{i=1}^{N-2}}{2} tj + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$t(n) = C1 * N + C2 * (N-1) + C3 * \frac{N^2-N}{2} + C4 * \frac{N^2-3*N+2}{2} + C5 * \frac{N^2-7*N+10}{4} + C6 * (N-1) + C7 * (N-1) + C8 * (N-1)$

$t(n) = N^2 * (\frac{C3}{2} + \frac{C4}{2} + \frac{C5}{2}) + N * (C1 + C2 - \frac{C3}{2} - \frac{7*C4}{4} - \frac{3*C5}{2} + C6 + C7 + C8) + (-C2 + \frac{2*C4}{2} + \frac{10*C5}{4} - C6 - C7 - C8)$

4.4 *Shell Sort*

De acordo com De Souza(2016), varios autores tentaram calcular a complexidade do *shell sort*, porem, mesmo após 60 anos não existe um consenso quanto a esse caso. A maior parte deles encontraram complexidades diferentes, e poucos conseguiram calcular o pior caso. Então, apesar de sua eficiencia o algoritmo não possui uma complexidade definida.

4.5 *Merge Sort*

Para a analise do *merge sort* primeiro devemos considerar a complexidade do merge, como ele é um algoritmo que une os vetores podemos considerar que ele é executado em $\Theta(n)$ quando está intercalando n elementos. Agora que definimos sua complexidade podemos analisar o *merge sort* representado na figura 51.

```
void merge_sort(int vetor[], int p, int r){
    if(p < r){
        int q = (p+r)/2;
        merge_sort(vetor, p, q);
        merge_sort(vetor, q+1, r);
        merge(vetor, p, q, r);
    }
}
```

Figura 51: Algoritmo *merge sort* em C++

Criando uma tabela onde cada linha possui um custo podemos calcular o custo de cada uma baseado em um número variavel de execuções.

Linha	Codigo	Custo	Número de Execuções
1	if(p < r)	C1	1
2	int q = (p+r)/2	C2	1
3	merge$_{s}ort(vetor, p, q)$	C3	N/2
4	merge$_{s}ort(vetor, q+1, r)$	C4	N/2
5	merge(vetor, p, q, r)	C5	N

Com isso podemos chegar em $T(N) = 2 * T(N/2) + N$ que podemos calcular usando recorrência.

$T(N) = 2 * T(N/2) + N$

$N = 2^K; K = \log_2 K$

$T(K) = 2 * T(2^(K-1)) + 2^K$

$T(K-1) = 2 * T(2^(K-2)) + 2^(K-1)$

$T(K-2) = 2 * T(2^(K-3)) + 2^(K-2)$

$T(K) = 2 * T(2^(K-1)) + 2^K$

$=> 2 * [2 * T(2^(K-2)) + 2^(K-1)] + 2^K$

$=> 2^2 * [2 * T(2^(K-3)) + 2^(K-2)] + 2 * 2^K$

$T(K) = 2^K * T(2^0) + (K-1) * 2^K$

$=> 2^K + (K-1) * 2^K$

$=> K * 2^K$

$T(N) = N * \log_2 N$

O consumo de tempo do *merge sort* é $O(N * log_2 N)$.

4.6 *Quick Sort*

Assim como o algoritmo *merge*, o particiona também possui tempo $\Theta(n)$, sabendo disso podemos calcular o tempo para cada caso.

Pior Caso: O pior caso, ao contrario da maioria dos algoritmos, se dará quando o vetor já estiver ordenado, pois, nesta situação a escolha do pivo vai ser sempre a pior se considera-lo como o primeiro elemento do intervalo, sendo assim a cada execução só será ordenado um elemento por vez. Neste caso o tempo de execução será dado por $T(N) = T(N-1) + N$.

$T(N) = T(N-1) + N$

$T(K) = T(K-1) + K$
$T(K-1) = T(K-2) + K - 1$
$T(K-2) = T(K-3) + K - 2$

$T(K) = T(K-1) + K$
$=> [T(K-2) + K - 1] + K$
$=> [T(K-1) + K - 2] + K - 1 + K$

$T(K) = 1 + 2 + ... + (K-2) + (K-1) + K$
$=> \frac{N*(1+N)}{2}$
$=> \frac{N^2+N}{2}$
Como pode ser visto o pior caso é representado por uma função quadratica.

Melhor Caso: Acontece quando as partições tem sempre o mesmo tamanho, significando que cada chamada recursiva do algoritmo tem custo N/2, e como também sabemos o custo do particiona podemos concluir que o melhor caso é representado por $T(N) = 2T(N/2) + N$

$T(N) = 2 * T(N/2) + N$

$N = 2^K; K = \log_2 K$

$T(K) = 2 * T(2^(K-1)) + 2^K$

$T(K-1) = 2 * T(2^(K-2)) + 2^(K-1)$

$T(K-2) = 2 * T(2^(K-3)) + 2^(K-2)$

$T(K) = 2 * T(2^(K-1)) + 2^K$

$=> 2 * [2 * T(2^(K-2)) + 2^(K-1)] + 2^K$

$=> 2^2 * [2 * T(2^(K-3)) + 2^(K-2)] + 2 * 2^K$

$T(K) = 2^K * T(2^0) + (K-1) * 2^K$

$=> 2^K + (K-1) * 2^K$

$=> K * 2^K$

$T(N) = N * \log_2 N$

A complexidade do melhor caso do *quick sort* é $O(N * log_2 N)$.

Caso Medio: Neste caso, de acordo com Sedgwick e Elasolet(1996, pag 17) o número de comparações é T(N) = 1,386*N*$\log_2 N$ - 0,846*N, o que significa que em média o tempo T(N) = $O(N * log_2 N)$.

4.7 *Heap Sort*

Para calcular a complexidade do *heap sort*, primeiro devemos saber a complexidade do *min heapify* e do *build min heap*. Como no *min heapify* a etapa de divisão ocorre com a determinação do maior entre o nó pai e seu filho da direita e da esquerda a complexidade será O(1), pois são feitas 2 comparações e não tem etapa de combinação, porém, no seu pior caso, onde o último nível do heap tem a metade completa, o tempo de execução pode ser descrito pela recorrencia T(n) T(2n/3) + O(1) que resulta em $O(\log n)$. Como alternativa podemos caracterizar o tempo de execução do *build min heap* como O(n). Com isso já podemos calcular a complexidade do codigo da figura 52.

```cpp
void heapSort(vector<int> &vetor) {
    build_min_heap(vetor);
    int tamanho = vetor.size() - 1;
    for (int i = vetor.size() - 1; i > 0; i--) {
        swap(vetor[0], vetor[i]);
        tamanho--;
        min_heapify(vetor, tamanho, 0);
    }
}
```

Figura 52: Código do *heap sort* em C++

Primeiramente é necessario definir os custos das linhas, que sera representado por uma variavel C para cada linha de codigo executavel, em seguida precisamos dos números de execução que será calculado com base no número de execuções da primeira linha como pode ser visto na tabela a seguir.

Linha	Codigo	Custo	Número de Execuções
1	$build_min_heap(vetor);$	C1	1
2	int tamanho = vetor.size() - 1;	C2	1
3	for (int i = vetor.size() - 1; i ¿ 0; i–)	C3	N
4	swap(vetor[0], vetor[i]);	C4	N-1
5	tamanho–;	C5	N-1
6	$min_heapify(vetor, tamanho, 0);$	C6	N-1

$T(n) = C1 + C2 + N * C3 + (N - 1) * C4 + (N - 1) * C5 + (N - 1) * C6$

$T(n) = N + C2 + N * C3 + (N - 1) * C4 + (N - 1) * C5 + (N - 1) * \log n$

$T(n) = N + C2 + N * C3 + (N - 1) * C4 + (N - 1) * C5 + N * \log n - \log n$

$T(n) = N(1 + C3 + C4 + C5 + \log n) + (C2 - C4 - C5 - \log n)$

A partir disso podemos ver que o algoritmo possui uma complexidade O($N * \log n$), essa complexidade é considerada para todos os seus casos, apesar de que o melhor caso, onde todos os elementos são iguais, possuir tempo O(n), por ser um caso pouco pratico não é considerado. O medio caso é quando o vetor está aleatoriamente ordenado, e o pior caso quando ele está em ordem decrescente.

5 TABELA E GRÁFICO

5.1 *Insertion Sort*

A fim de comparação, podemos analisar as entradas no pior, melhor e médio caso com base no tamanho do vetor de entrada, como demonstrado na figura53.

	10	100	1000	10000	100000	1000000
CRESCENTE	0	0	0	0	0	0
ALEATORIO	0	0	0	0.091	9.388	630.958
DECRESCENTE	0	0	0.002	0.415	22.643	1739.25

Figura 53: Tabela de tempo em relação a entrada do algoritmo *Insertion Sort*

As ordenações não diferem muito até a entrada de tamanho 10.000, mas a partir dai a diferença cresce exponencialmente nos caso médio e no pior caso, enquanto no melhor caso o tempo de execução é 0. É possível visualizar essa diferença no gráfico da figura 54.

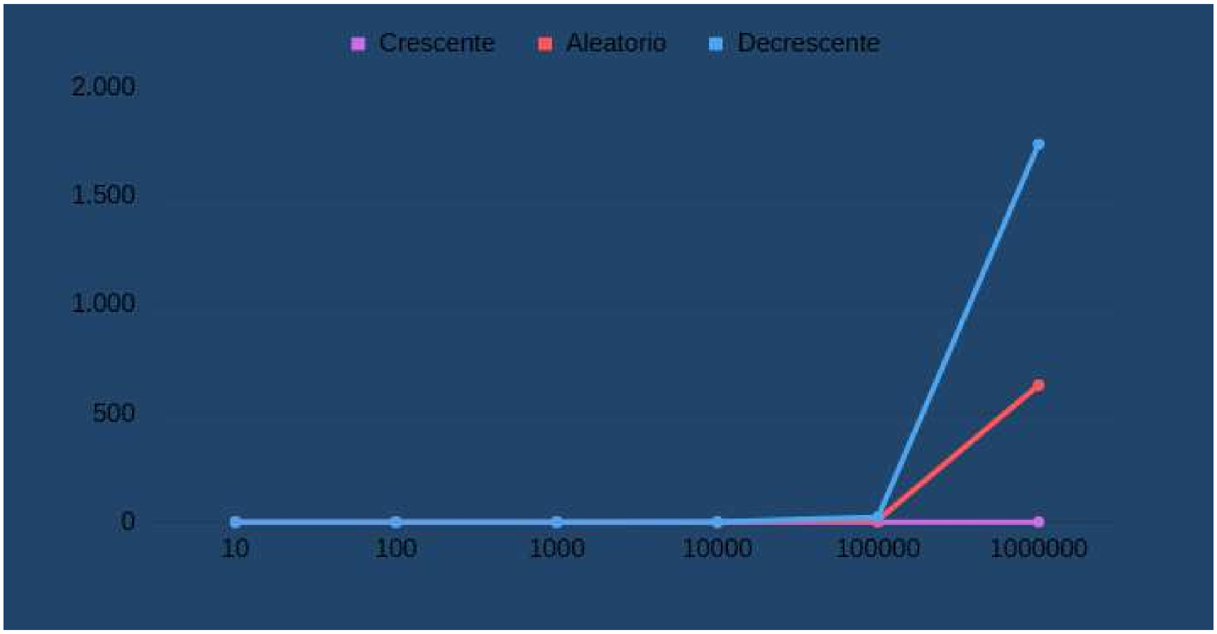


Figura 54: Gráfico de tempo por entrada do algoritmo *Insertion Sort*

5.2 *Bubble Sort*

Analisando o tempo de ordenação para cada entrada do *bubble sort* da figura 55, podemos concluir que se trata de um algoritmo com eficiencia extremamente baixa para vetores com muitos elementos, sendo superior ao *insertion sort* apenas na entrada de um milhão de elementos aleatorios.

	10	100	1000	10000	100000	1000000
CRESCENTE	0	0	0.002	0.142	23.499	1293.22
ALEATORIO	0	0.001	0.002	0.23	33.526	5118.69
DECRESCENTE	0	0	0.002	0.405	29.757	4326.25

Figura 55: Tabela de tempo em relação ao tamanho da entrada do algoritmo *bubble sort*

Através do gráfico da figura 56 fica claro que o tempo de ordenação cresce de forma extremamente rapida a partir de entradas de tamanho 100 mil, e que mesmo para um vetor ordenado o tempo gasto permanece alto.

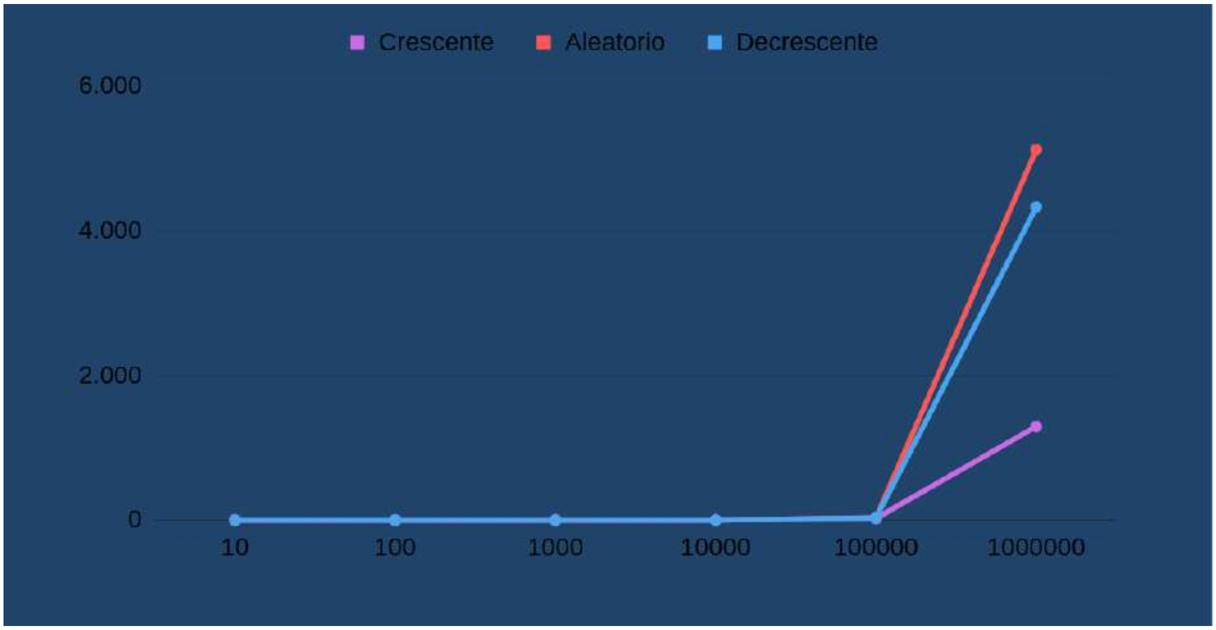


Figura 56: Gráfico de tempo por entrada do algoritmo *bubble sort*

5.3 *Selection Sort*

Pela analise dos tempos em relação ao tamanho da entrada, podemos perceber que o algoritmo *selection sort* apresenta um desempenho relativamente bom em suas ordenações, como representado na tabela 57

	10	100	1000	10000	100000	1000000
CRESCENTE	0	0	0.006	0.111	11.388	1159.21
ALEATORIO	0	0	0.001	0.111	11.442	1409.70
DECRESCENTE	0	0	0.005	0.095	9.969	1323.73

Figura 57: Tabela de tempo em relação a entrada do algoritmo *Selection Sort*

Fazendo uma comparação agora atraves do grafico da figura 58, podemos ver que a variação de tempo entre as entradas é significativamente menor quando comparado aos algoritmos anteriores.

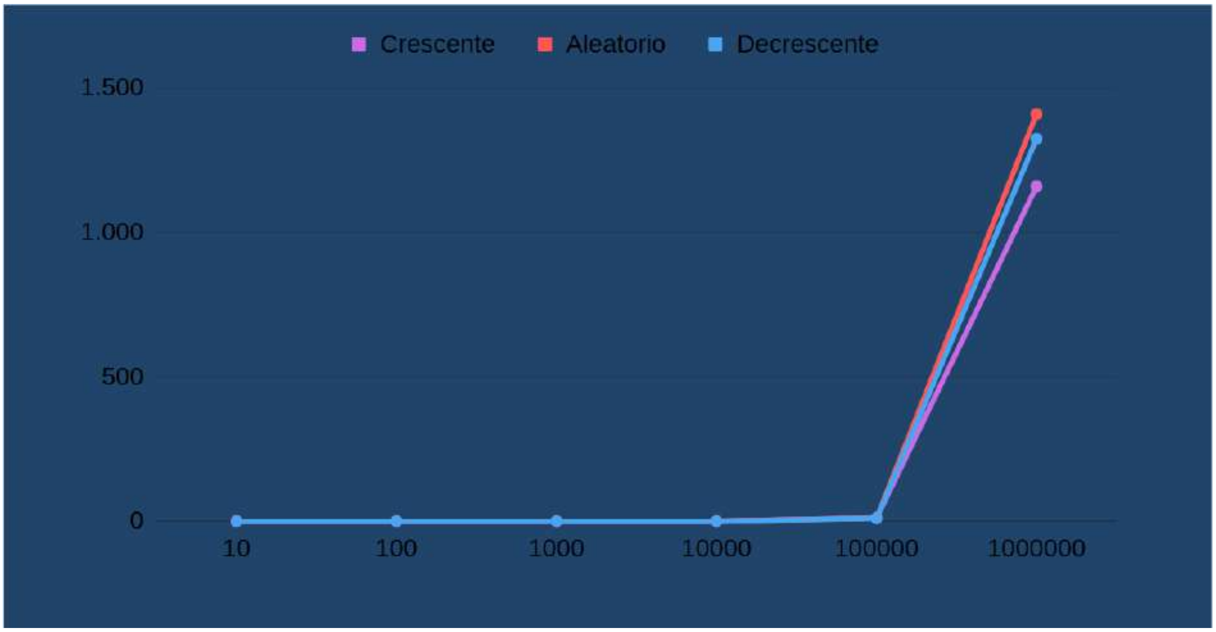


Figura 58: Gráfico de tempo por entrada do algoritmo *Selection Sort*

5.4 *Shell Sort*

O *shell sort* é um algoritmo que se destaca dentre todos analisados até agora, ele possui um tempo incrivelmente bom para todos os tipos e tamanhos de entrada, ordenando vetores em menos de um segundo, conforme a figura 59

	10	100	1000	10000	100000	1000000
CRESCENTE	0	0	0	0.001	0.011	0.082
ALEATORIO	0	0	0	0.004	0.036	0.565
DECRESCENTE	0	0	0	0.001	0.009	0.166

Figura 59: Tabela de tempo em relação a entrada do algoritmo *Shell Sort*

Seu grafico é semelhante ao do *selection sort*, onde as entradas e tamanhos possuem uma variação de tempo muito pequena, mas, no caso do *shell sort* essa variação é de milesimos.

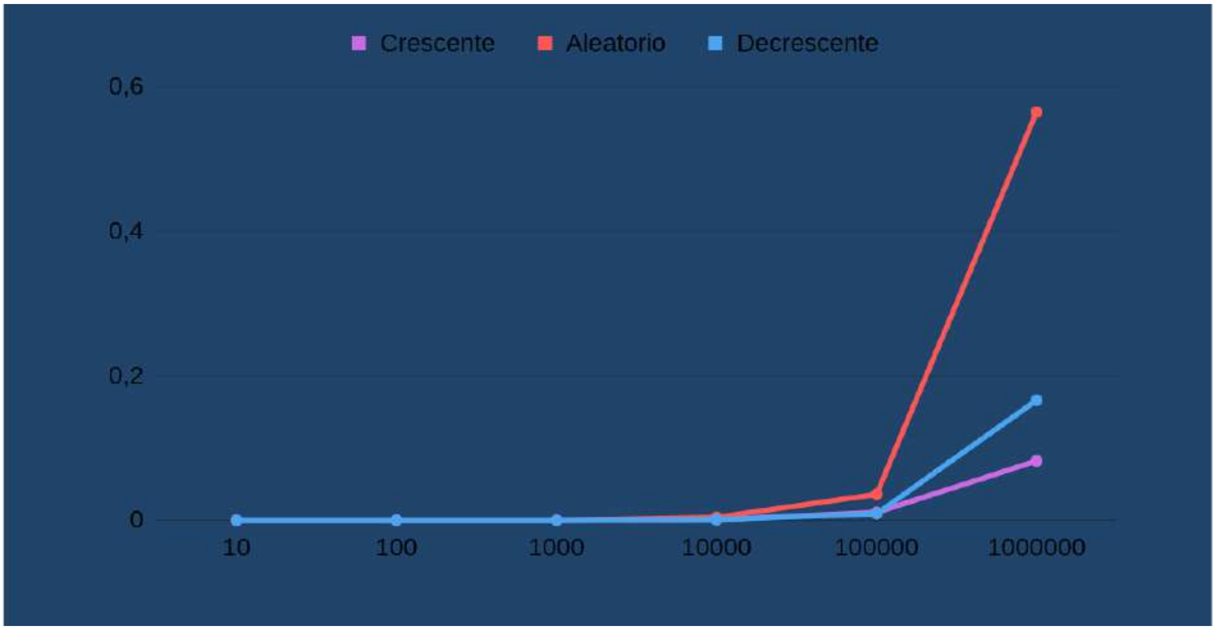


Figura 60: Gráfico de tempo por entrada do algoritmo *Shell Sort*

5.5 *Merge Sort*

O método de divisão e conquista presente no *merge sort* apresenta uma grande eficiencia na ordenação de vetores, como pode ser visto na tabela 61, apesar de não ser tão eficiente quanto o *shell sort*.

	10	100	1000	10000	100000	1000000
CRESCENTE	0	0	0	0.002	0.041	0.284
ALEATORIO	0	0	0	0.002	0.026	0.336
DECRESCENTE	0	0	0	0.001	0.01	0.257

Figura 61: Tabela de tempo em relação a entrada do algoritmo *Merge Sort*

Dentre todos os algoritimos apresentados, esse possui a menor diferença entre os casos, sendo menor que 0.1 segundo em entradas de tamanho um milhão.

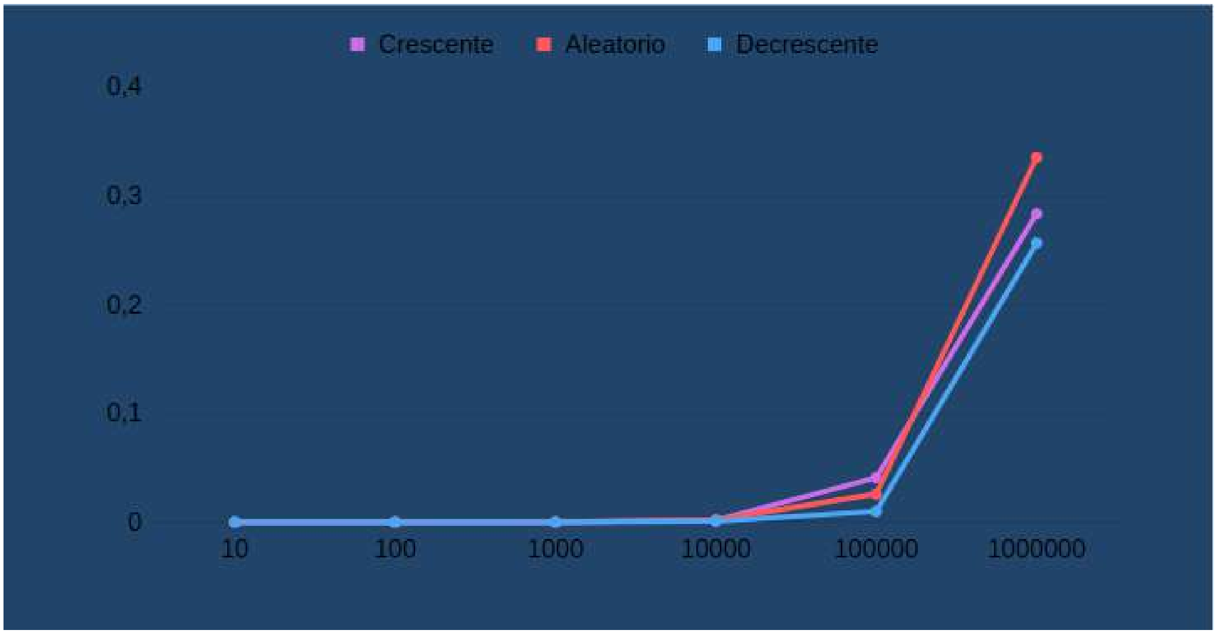


Figura 62: Gráfico de tempo por entrada do algoritmo *Merge Sort*

5.6 *Quick Sort*

Assim como o *merge sort*, o *quick sort* utiliza o método da divisão e conquista, pórem, ele funciona de forma diferente dos demais algoritimos. Na figura 63 podemos ver um comportamento bem diferente, onde ao enves de apresentar uma curva de tempo linear, apresenta uma curva irregular.

	10	100	1000	10000	100000	1000000
CRESCENTE	1.018	1.363	1.029	1.407	1.861	2.753
ALEATORIO	2.714	1.003	0.771	1.225	1.006	1.155
DECRESCENTE	0.812	0.782	1.251	1.232	1.772	2.635

Figura 63: Tabela de tempo em relação a entrada do algoritmo *Quick Sort*

Esse padrão irregular pode ser melhor visualizado no grafico da figura 64.

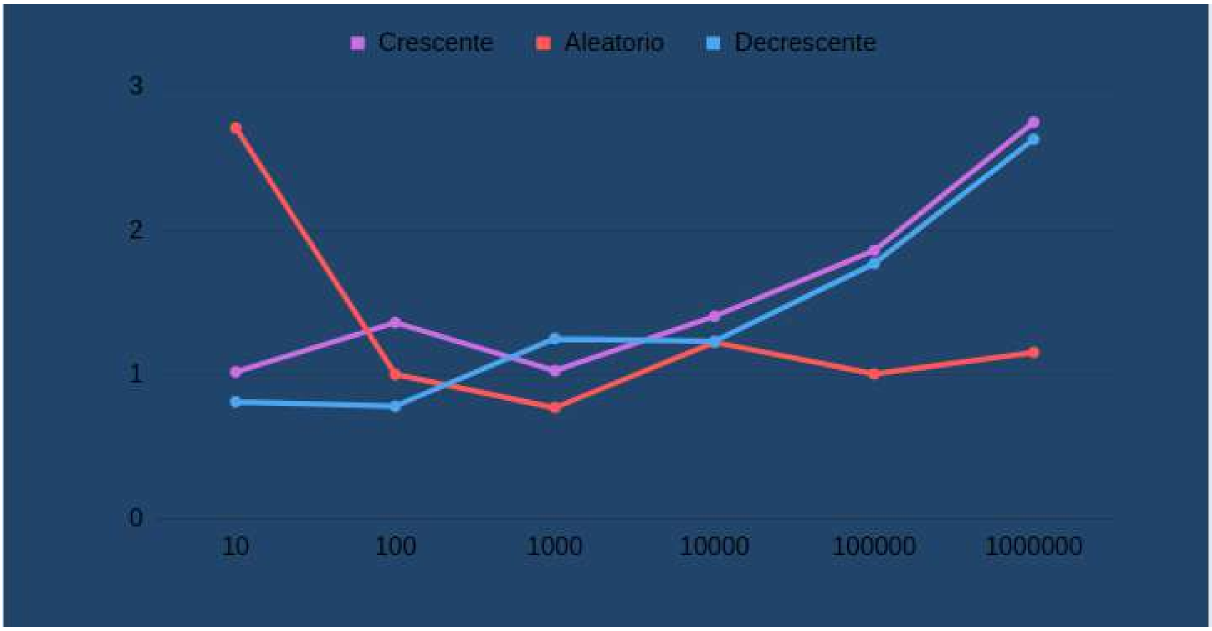


Figura 64: Grafico de tempo por entrada do algoritmo *Quick Sort*

5.7 *Heap Sort*

Análisando o tempo de execução do *heap sort* com vetores que variam de 10 a 1000000 elementos chegamos a tabela da figura 65.

	10	100	1000	10000	100000	1000000
CRESCENTE	0.777	0.744	0.868	0.789	0.833	1.537
ALEATORIO	1.268	0.632	0.973	1.203	0.656	1.6
DECRESCENTE	0.675	0.685	1.740	0.648	0.750	1.508

Figura 65: Tabela de tempo em relaçãoo a entrada do algoritmo *heap sort*

podemos ver a partir dos resultados que o tempo de execução quando o vetor era crescente, decrescente e aleatorio são bem semelhantes, possuindo uma variação de milesimos, exceto no caso de um vetor de 1000 elementos ordenado de forma decrescente, que resultou em um gasto de tempo maior que o esperado. Isso pode ser melhor observado na tabéla da figura 66.

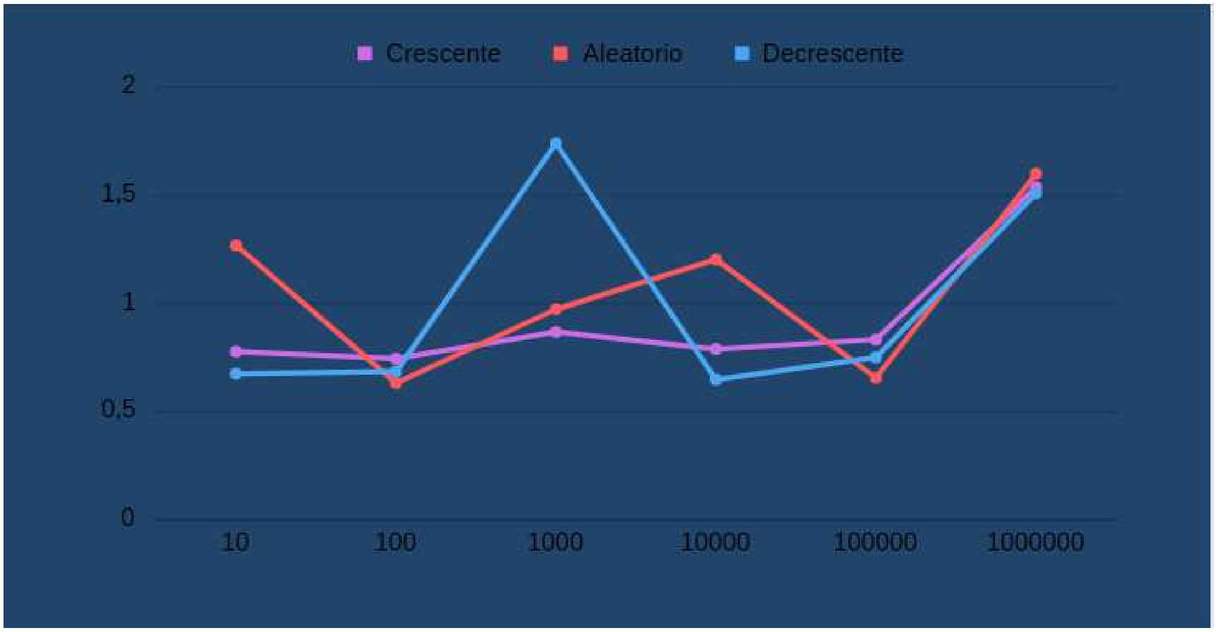


Figura 66: Gráfico de tempo por entrada do algoritmo *heap sort*

5.8 *Comparativo Geral*

Agora, analisando lado a lado os resultados obtidos por cada algoritmo de ordenação, podemos ver claramente os algoritmos que possuem maior gasto para ordenar vetores em ordem crescente, decrescente e aleatório, conforme, respectivamente, as figuras **??**, **??** e **??**.

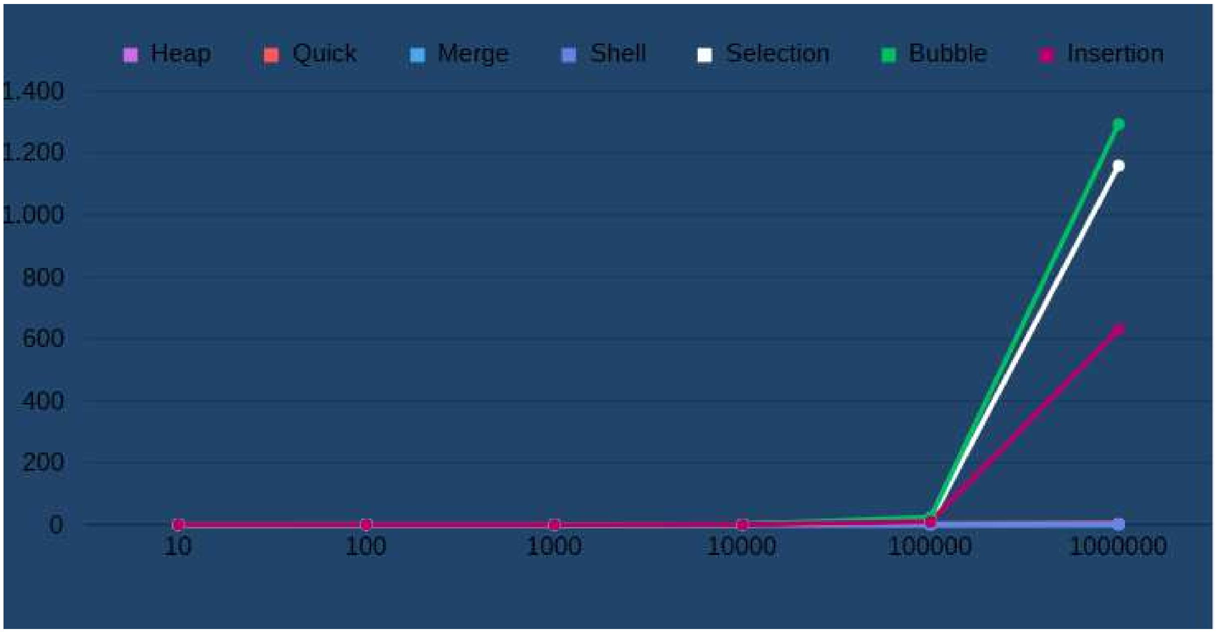


Figura 67: Gráfico de tempo por entrada dos algoritmos para vetores crescentes

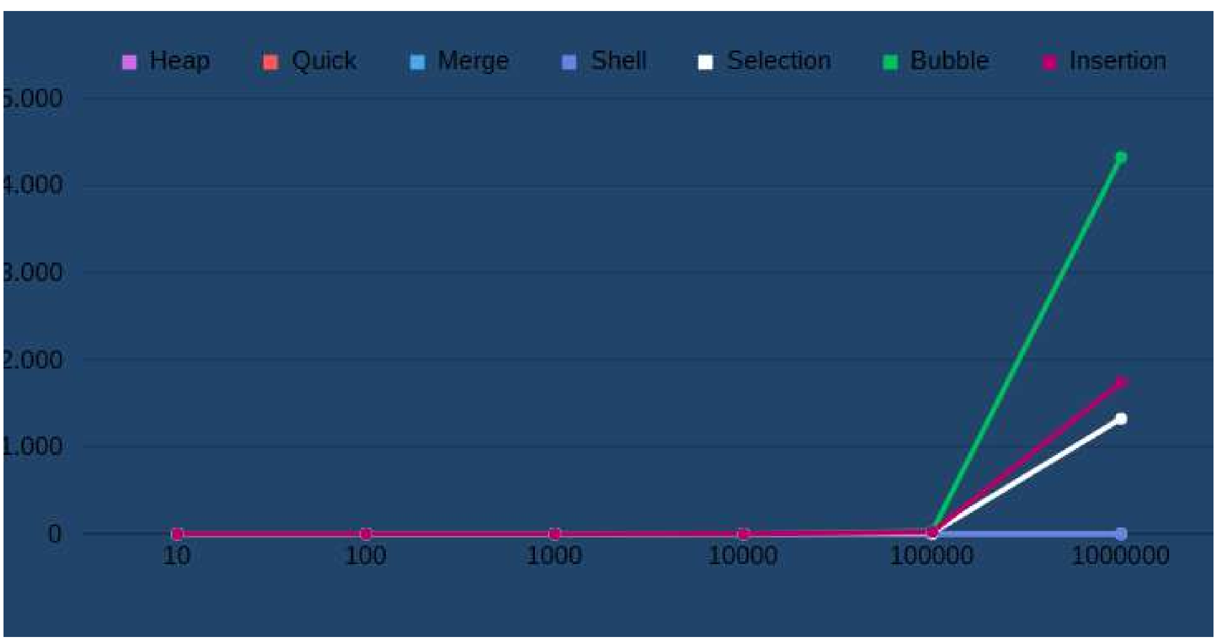


Figura 68: Gráfico de tempo por entrada dos algoritmos para vetores decrescentes

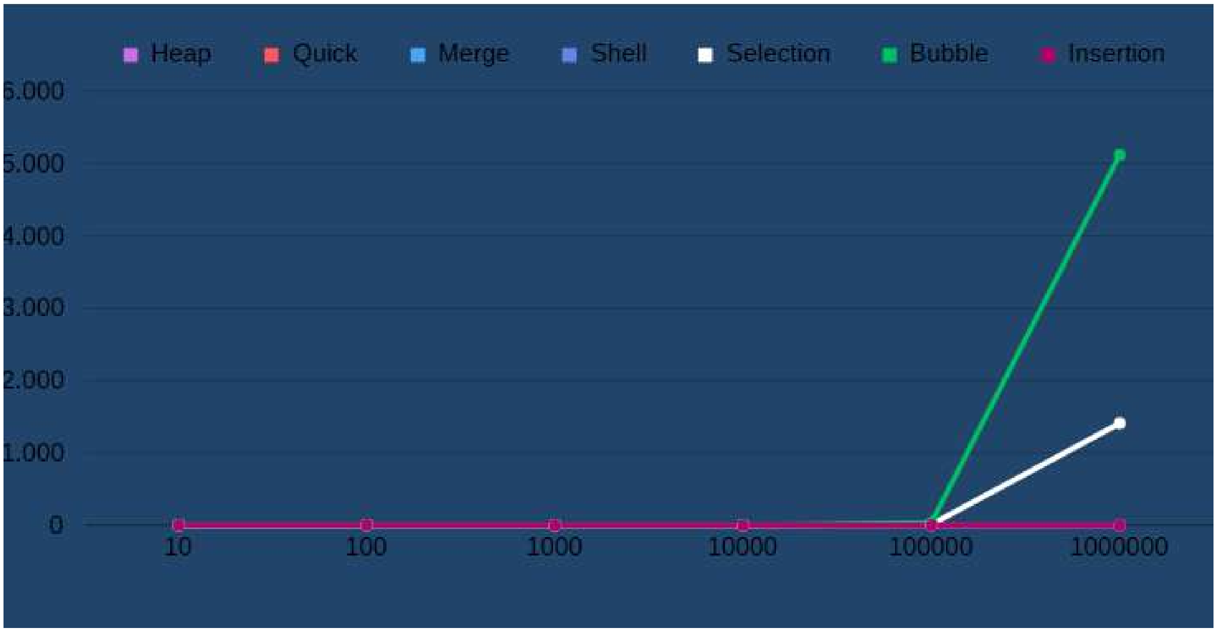


Figura 69: Gráfico de tempo por entrada dos algoritmos para vetores aleatórios

6 CONCLUSÃO

Portanto, considerando os estudos e os testes feitos no algoritmo *insertion sort*, podemos concluir que ele é muito eficiente quando se trata de ordenar vetores até o tamanho 100.000, porem, a partir disso o tempo de execução aumenta exponencialmente.

No caso do *bubble sort* podemos perceber que ele é eficiente quando se tem poucos números, porém, quando é necessario ordenar um vetor com muitos elementos ele é muito pouco eficiente.

O *selection sort* é melhor que o *bubble sort*, mas ele também possui uma eficiencia decadente quando se trata de vetores muito grandes.

O melhor algoritmo de complexidade $O(N^2)$ é definitivamente o *shell sort*, sendo incomparavel aos anteriores quando se trata de eficiencia, sendo extremamente eficiente para vetores pequenos e grandes.

Comparando o *merge sort*, *quick sort*, e o *shell sort* podemos chegar a rapida conclusão de que o *shell sort* se destaca, possuindo um tempo inferior ao em entradas crescentes e decrescentes, sendo inferior ao *merge sort* no tempo de entradas aleatorias. O *quick sort*, apesar de possuir um tempo muito bom, quando comparado aos dois outros é visivel que possui um gasto de tempo muito maior.

O *heap sort* é um algoritmo que possui tempo de execução excelente, mas, quando comparado ao *shell sort* deixa a desejar, porem, o *heap sort* possui uma enorme utilidade quando se trata de filas de prioridade, se destacando muito neste quesito.

7 REFERÊNCIAS BIBLIOGRÁFICAS

Cormen, Thomas H [et al.]. *Algoritmos: Teoria e Prática.* tradução da segunda edição [americana] Vandenberg D. de Souza. - Rio de Janeiro : Elsevier, 2002 - Quarta Reimpressão.

De Souza, Raquel M.; Oliveira, Fabiano de S.; Pinto, Paulo Eustáquio D. Análise empírica do algoritmo Shellsort. In: Anais do I Encontro de Teoria da Computação. SBC, 2016. p. 903-906.