76B8 – ATIVIDADES PRÁTICAS SUPERVISIONADAS

Pedro Lorencini Favarão - N549645

Joao Victor Lorencini Favarão - N5496E2

Vitor Hugo Bonfadini de Souza - F197AI6

Victor Thauan de Andrade - N634CG0

SUMÁRIO

SÚMARIO – IMAGENS

Figura **Página**

1. Objetivo e motivação.

Essa atividade tem como objetivo aprofundar os conhecimentos a respeito da matéria para com os alunos do curso e assim, prepara-los psicologicamente e profissionalmente para o mercado de trabalho que os mesmo irão encarar ou já encaram.

Coloca a prova o conhecimento dos alunos no quesito “codificação” em *C*, além do trabalho em equipe (fator que será comum na rotina dos mesmos). Além desses pontos importantes, os alunos são induzidos a realizar pesquisas nos fóruns da internet (ou outras fontes) para conseguir concluir a atividade em questão, influenciando indiretamente a habilidade de pesquisa deles.

O grupo responsável por esse relatório chegou à decisão de elaborar um programa que funcionaria para criar um determinado numero e ordena-lo, onde é possivel comparar o tempo de ordenação de cada sort. E partindo do ponto de que determinada linha de codigo aloca espaço na memoria, uma quantia considerável, saber qual codigo irá realizar o procedimento mais rapido e alocando menor memoria do sistema torna- se de certa forma necessária, evitando futuras frustações e complicações com os mesmos.

O codiano no mundo moderno faz com que nossos dias sejam cada vez mais agitados, a importancia de otimização do tempo faz com que as pessoas busquem novas formas de tecnologia para obter uma maior eficiencia, caso um usuario faça o mesmo trabalho de ordenação de dados por dia, é importante que essa ordenação seja feita da forma mais rapida possivel, para que o cliente consiga efetuar mais vezes o mesmo procedimento em um determinado tempo.

Alguns beneficios extras que são derivados pela a eficiencia de tempo no trabalho:

- Melhoria da qualidade do trabalho;
- Tempo livre para novos projetos;

Em resumo: o gerenciamento do tempo é praticamente uma obrigação para o meio profissional, devido a isso vem a importancia da estrutuação de dados.

2. Introdução

O primeiro passo é entender o que é a estrutura de dados: Estrutura de dados é uma estrutura organizada de dados na memória de um computador ou em qualquer dispositivo de armazenamento, de forma que os dados possam ser utilizados de forma correta.

É possivel encontrar essas estruturas em muitas aplicações no desenvolvimento de sistemas, sendo algumas altamente especializadas e utilizadas em tarefas específicas.

Utilizar as estruturas adequadas através de algoritmos, é possível trabalhar com uma grande quantidade de dados, como aplicações em bancos de dados ou serviços de busca.

Ao realizar uma estrutura de dados é necessario saber como realizar um determinado conjunto de operações básicas, como por exemplo: Inserção de dados, exclusão de dados, localizar um elemento, percorrer todos os itens constituintes da estrutura para visualização, classificar que se resume em colocar os itens de dados em uma determinada ordem (numérica, alfabética, etc.).

*Informações retiradas de: https://digitalinnovation.one/artigos/aprenda-o-que-sao-estrutura-de-dados-e-algoritmos-material-curso-dio

- Ponteiros

Um ponteiro é um tipo de variável que armazena um endereço, por exemplo: "p aponta para i", em termos mais simples, pode-se dizer que p é uma referencia à variavel i.

Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa. O código pode ter vários ponteiros espalhados por diversas partes do programa, “apontando” para a variável que contém o dado desejado. Caso o dado apontado seja alterado, não há problema algum, pois todas as partes do programa tem um ponteiro que aponta para o endereço onde reside o dado atualizado.

*Informações retiradas de: http://linguagemc.com.br/ponteiros-em-c/

Há varios tipos de ponteiros, sendo eles: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para registros, etc. É importante lembrar, para que seja utilizado, é necessario informar ao computador qual ponteiro esta se referindo.

- Vetores

O vetor é uma estrutura de dados indexada, que pode armazenar uma determinada quantidade de valores do mesmo tipo. Os dados armazenados em um vetor são chamados de itens do vetor.

Para localizar a posição de um item em um vetor é utilizado um número inteiro denominado índice do vetor.

A vantagem de utilização do vetor é a facilidade de manipular um grande conjunto de dados do mesmo tipo declarando-se apenas uma variável.

Quando é alocado um espaço para armazenar apenas um dado do tipo int, é comum utilizar ponteiros para alocação de vetores. Para isso, basta especificar o tamanho desse vetor no momento da alocação. Nos exemplos abaixo, apresenta-se a alocação de vetores com malloc e new. Após a alocação de uma área com vários elementos, ela pode ser acessada exatamente como se fosse um vetor.

- Casting

Casting ou tambem conhecido "Conversão" é um tipo de opações feita com objetivo de alterar o tipo de um determinado valor. Por Exemplo: 3.1415 / 3.0;

O resultado será numero fracionario, caso seja necessario apenas a parte inteira de uma divisção pode-se fazer a operação de Casting, dessa forma a parte fracionaria será desconsiderada e teremos apenas a parte inteira.

Uma divisão entre dois numeros inteiros irá gerar um resultado inteiro. A operação de casting pode ser feita, obetendo assim um resultado exato da divisão no tipo real.

Resumindo: Casting é uma conversão entre dois números podendo gerar um resultado inteiro ou de qualquer outra forma, dependendo da finalidade do codigo.

- Alocação de Memória

Na liguagem C, existem dois tipos de alocação de memória, sendo elas estatica e dinamica, cada uma possui suas caracteristicas, que podem ser beneficiadas dependendo o uso do programador, caso seja feita o mal uso, pode ocupar memoria sem necessidade.

- Estatica:

Na alocação estática de memória, os tipos de dados tem tamanho predefinido. Neste caso, o compilador vai alocar de forma automática o espaço de memória necessário. Sendo assim, dizemos que a alocação estática é feita em tempo de compilação.

Este tipo de alocação tende a desperdiçar recursos, já que nem sempre é possível determinar previamente qual é o espaço necessário para armazenar as informações. Quando não se conhece o espaço total necessário, a tendência é exagerar pois é melhor superdimensionar do que faltar espaço.

- Dinamica:

Na alocação dinâmica podemos alocar espaços durante a execução de um programa, ou seja, a alocação dinâmica é feita em tempo de execução.

Isto é bem interessante do ponto de vista do programador, pois permite que o espaço em memória seja alocado apenas quando necessário. Além disso, a alocação dinâmica permite aumentar ou até diminuir a quantidade de memória alocada.

Alocação dinamica possui funções que auxiliam nesse processo, os mais utilizados são:

sizeof: A função sizeof determina o número de bytes para um determinado tipo de dados. É interessante notar que o número de bytes reservados pode variar de acordo com o compilador utilizado.

malloc: A função malloc aloca um espaço de memória e retorna um ponteiro do tipo void para o início do espaço de memória alocado.

free: A função free libera o espaço de memória alocado.

- Função Clock

A função clock retorna o tempo de execução exato do momento em que ela foi chamada. Para encontrar o tempo de execução de um programa precisamos usar ela duas vezes, uma para capturar o tempo inicial e outra para capturar o tempo final da execução.

Se fizermos o tempo final menos o tempo inicial teremos o tempo de execução do programa em milissegundos. Dividindo esse valor pelo CLOCKS_PER_SEC teremos este valor em segundos, pois esta constante tem o valor de um milhão (1000000). Para obter o valor em milissegundos, pode-se dividir o CLOCKS_PER_SEC por mil (1000).

Lembrando que: a variável que irá armazenar o valor do tempo da função clock deve ser do tipo clock_t.

Para fazer uso da função que irá retornar o tempo de execução de um programa é necessário chamar a biblioteca 'time.h'.

Para chamar essa biblioteca basta por no cabeçalho do seu programa: #include <time.h>

3. Plano de desenvolvimento da aplicação

O projeto inicial era realizar a leitura de um arquivo do tipo txt, esse arquivo teria um numero pré definido de caracteres e totalmente seria desordenado. Ao iniciar a aplicação, o arquivo de texto seria lido, o aplicativo realizaria a ordenação, calcularia o tempo que levou para realizar todas as ordenações e por fim exibiria para o usuario em forma de print, após isso seria salvo no mesmo arquivo de leitura, porém totalmente ordenado; Sendo assim, o arquivo entraria com numeros aleatorios e desordenado e sairia totalmente ordenado.

Ao desenvolver do projeto, foi identificado varias falhas na arquitetura inicial, a primeira é que, um arquivo com poucos caracteres seria facilmente ordenado até pelo mais lento dos sorts, o que não daria nenhum tempo de execução. Para que desse algum retorno na excussão do programa, o arquivo teria que ter mais de 10.000 (dez mil) caracteres, o que seria inviavel, criar um arquivo de texto todas as vezes que fosse rodar o aplicativo, contendo mais de dez mil numeros aleatoriamente informados.

Mesmo que o problema do arquivo fosse resolvido, seria necessario criar um arquivo txt toda vez que fosse rodar o programa, ja que o programa lia o arquivo, ordenava e salvava.

Para solucionar o seguinte impecilio, foi pensado em uma forma de criar esses vetores de forma automatica, sem a necessidade de criar um arquivo com mil ou mais numeros, ja que seria implementada uma forma automatica de criar os vetores, foi considerado a possibilidade de disponibilizar para o proprio usuario o tamanho que deseja criar, assim, o tamanho do vetor seria informado pelo usuario e a sua criação seria de forma automatica; Como não haveria mais arquivo para ser lido, indiretamente o plano para salvar em um arquivo de texto foi deixado de lado.

Com o projeto pronto, deu inicio a sua execussão, o primeiro passo foi entender melhor sobre as funções de ordenação e alocação de memoria, foi feita uma pesquisa para uma das funções abaixo:

- BubbleSort
- QuickSort
- InsertionSort
- BinaryInsertionSort
- SelectionSort
- HeapSort
- MergeSort
- BucketSort

Então foi dado inicio a estrutura do programa, ao montarmos os tres primeiros algoritmos de ordenação, nos deparamos com um problema, como o vetor criado era uma variavel global, depois que o primeiro sort realiza a ordenação, esse vetor passava pelas outras funções ja ordenado, ou seja, não era feito ordenação pelas outras operações, apenas era lido e apresentado o tempo que aquela função levou para ler, ja que não era necessario ordenar.

Depois de pesquisar em foruns e com auxilio da professora, foi criado um segundo vetor, onde o primeiro armazena todo os dados desordenados e para que seja feita a ordenação, esses dados são passados para o segundo vetor, ordenados e calculado o tempo de execução, após isso esse vetor é limpo e na proxima função é jogado os dados desordenados novamente.

Após ajustar o codigo, foi inseridos os demais codigos de ordenação, após os ajustes, enfratamos um novo desafio, o Binary Insertion Sort, como seu algoritmo funciona baseado em Binary Seach, foi necessario realizar uma longa pesquisa sobre o conceito para entender como funciona, e ai sim aplicar, seguidos de alguns erros de sintaxe, porém resolvidos, finalmente tinhamos o projeto completo. Com tudo pronto, foi realizado alguns ajustes e por fim finalmente estava finalizado, o aplicativo era capaz de ler a entrada do usuario, criar o vetor com valores aleatorios, salvar em dois vetores diferentes e chamar apenas uma na hora de realizar a ordenação.

4. Projeto (estrutura e módulos que serão abordados)

Ao falarmos de ordenação de dados, nos vem à mente dados ordenados de alguma forma – possivelmente por causa da palavra ordenação -. Essa linha de pensamento não está incorreta, mas vale ressaltar que há várias maneiras de ordenar algum tipo de dado, entre os mais populares temos:

4.1 Insertion Sort

Algoritmo simples e eficiente **quando aplicado em pequenas listas**. Neste algoritmo a lista é percorrida da esquerda para a direita, à medida que avança vai deixando os elementos mais à esquerda ordenados. O algoritmo funciona da mesma forma que as pessoas usam para ordenar cartas em um jogo de baralho como o pôquer.

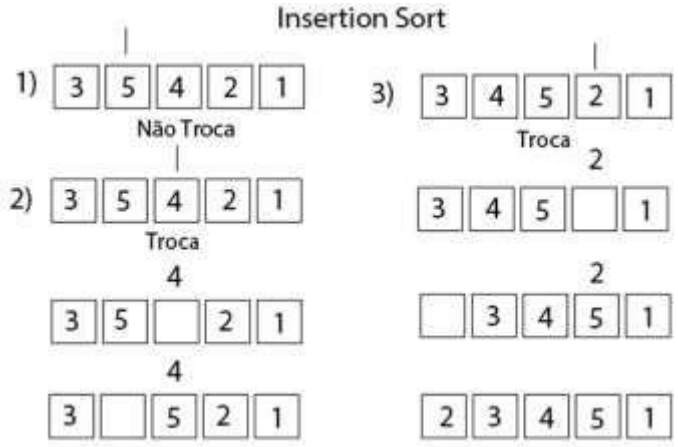

Figura 1 - Exemplo de Insertion Sort

- Neste passo é verificado se o 5 é menor que o 3, como essa condição é **falsa**, então não há troca após isso, é verificado se o quatro é menor que o 5 e o 3, ele só é menor que o 5, então **os dois trocam de posição**.
- É verificado se o 2 é menor que o 5, 4 e o 3, como ele é menor que 3, então o 5 passa a ocupar a posição do 2, o 4 ocupa a posição do 5 e o 3 ocupa a posição do 4, assim a posição do 3 fica vazia e o 2 passa para essa posição.
- O mesmo processo de comparação acontece com o número 1, após esse processo o vetor fica ordenado. (Bruno – Algoritmos de ordenação)

4.2. *Bubble Sort*

É o algoritmo **mais simples, mas o menos eficiente**. Neste algoritmo cada elemento da posição i será comparado com o elemento da posição i + 1, ou seja, um elemento da posição 2 será comparado com o elemento da posição 3. Caso o elemento da posição 2 for maior que o da posição 3, eles trocam de lugar e assim sucessivamente. Por causa dessa forma de execução, o vetor terá que ser percorrido quantas vezes que for necessária, tornando o algoritmo **ineficiente para listas muito grandes**.

Figura 2 - Exemplo de Bubble Sort

- É verificado se o 3 é maior que 5, por essa condição ser **falsa**, **não há troca**.
- É verificado se o 5 é maior que 1, por essa condição ser **verdadeira**, **há uma troca**.

- É verificado se o 5 é maior que 2, por essa condição ser **verdadeira**, **há uma troca**.
- É verificado se o 5 é maior que 4, por essa condição ser **verdadeira**, **há uma troca**.
- O método retorna ao início do vetor realizando os mesmos processos de comparações, isso é feito até que o vetor esteja ordenado.

(Bruno – Algoritmos de ordenação)

4.3 Binary Insertion Sort

O Binary Insertion Sort é um algoritmo de classificação semelhante ao Insertion Sort, mas em vez de usar a pesquisa linear para encontrar a posição onde o elemento deve ser inserido, usamos pesquisa binária. Assim, reduzimos o número de comparações para inserir um elemento de O(N) para O(log N).

É um algoritmo adaptativo, o que significa que ele funciona mais rápido quando a matriz dada já está substancialmente classificada, ou seja, a posição atual do elemento está perto de sua posição real na matriz classificada.

É um algoritmo de classificação estável - os elementos com os mesmos valores aparecem na mesma ordem na matriz final como estavam na matriz inicial.

(Interview Kickstart - Binary Insertion Sort)

4.4 Quick Sort

O algoritmo **mais eficiente na ordenação por comparação**. Nele escolhe-se um elemento chamado de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada.

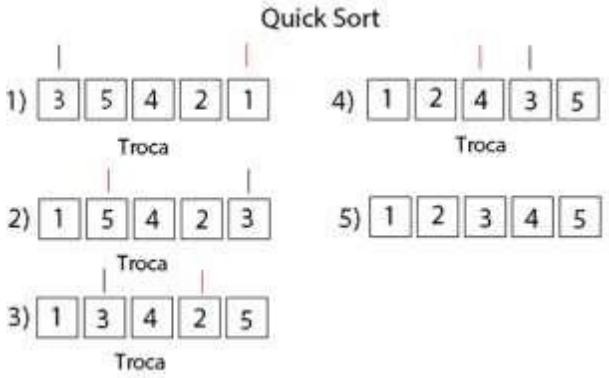

Figura 3 - Exemplo de Quick Sort

- O número 3 foi escolhido como pivô, nesse passo é procurado à sua direita um número menor que ele para ser passado para a sua esquerda. O primeiro número menor encontrado foi o 1, então **eles trocam de lugar**.
- Agora é procurado um número à sua esquerda que seja maior que ele, o primeiro número maior encontrado foi o 5, portanto **eles trocam de lugar**.
- O mesmo processo do passo 1 acontece, o número 2 foi o menor número encontrado, **eles trocam de lugar**.
- O mesmo processo do passo 2 acontece, o número 4 é o maior número encontrado, **eles trocam de lugar**.
- O vetor desse exemplo é um vetor pequeno, portanto ele já foi ordenado, mas se fosse um vetor grande, ele seria dividido e recursivamente aconteceria omesmo processo de escolha de um pivô e comparações.

(Bruno – Algoritmos de ordenação)

4.5 Merge Sort

Exemplo de algoritmo de ordenação por comparação do tipo **dividir-para-conquistar**. Sua ideia básica consiste em *Dividir* (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e *Conquistar* (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).

Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas. (Não autenticado – Merge Sort)

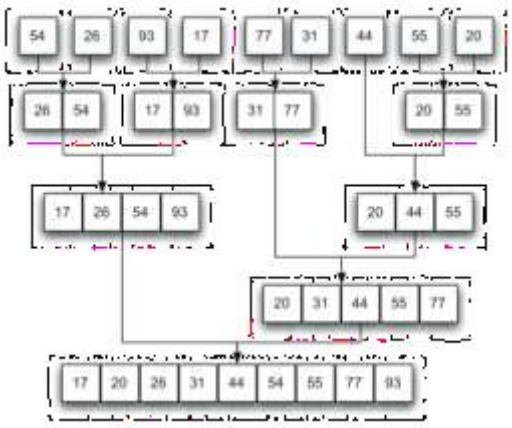

Figura 4 - Exemplo de Merge Sort (Brad Miller & David Ranum- O Merge Sort)

4.6 Heap Sort

Utiliza uma estrutura de dados chamada Heap* para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap*, na ordem desejada, sendo essencial que a propriedade max-heap seja mantida. Essa propriedade garante que o valor de todos os nós são menores que os de seus respectivos pais.

***Heap**: Pode ser representada como uma árvore (árvore binária com propriedades especiais) ou como um vetor. (Gleyser Guimarães – O AlgoritmoHeapSort).

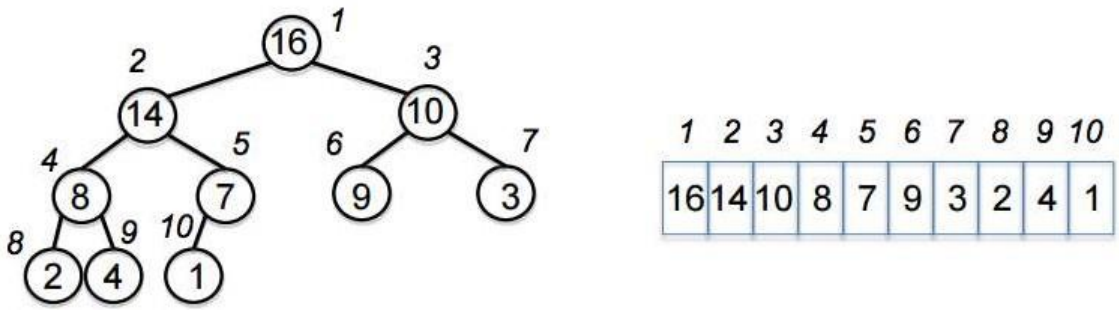

Figura 5 - Exemplo de Heap Sort (Brilliant - Heap Sort)

4.7 Bucket Sort

O algoritmo é baseado na ideia do uso de chaves como indices em um arranjo B de Buckets. Tal arranjo, possui entradas no intervalo de 0 a [N−1], onde N representa a quantidadede chaves. Cada posição de B em uma lista de itens. Por exemplo, o elemento f armazenado em B[f].

O Bucket Sort funciona da seguinte maneira: seja S a sequencia que deseja-se ordenar. Cada elemento de S é inserido em seu Bucket. Em seguida, ordena-se os Buckets e o conteudo é devolvido em S.

(Domingos Lacerda - Algoritmos de OrdenaçãoAlgoritmos de Ordenação)

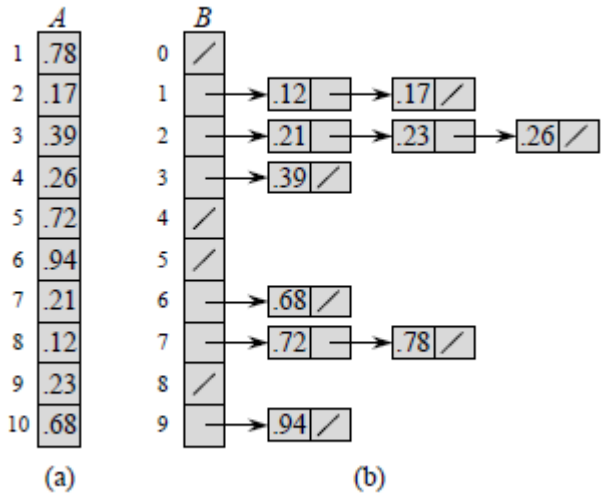

Figura 6 - Exemplo de Bucket Sort

4.8 Selection Sort

Baseado em se **passar sempre o menor valor do vetor para a primeira posição** (ou o maior dependendo da ordem requerida), depois o segundo menor valor para a segunda posição e assim sucessivamente, até os últimos dois elementos.

Neste algoritmo de ordenação é escolhido um número a partir do primeiro, este número escolhido é comparado com os números a partir da sua direita, quando encontrado um número menor, o número escolhido ocupa a posição do menor número encontrado. Este número encontrado será o próximo número escolhido, caso não for encontrado nenhum número menor que este escolhido, ele é colocado na posição do primeiro número escolhido, e o próximo número à sua direita vai ser o escolhido para fazer as comparações. É repetido esse processo até que a listaesteja ordenada.

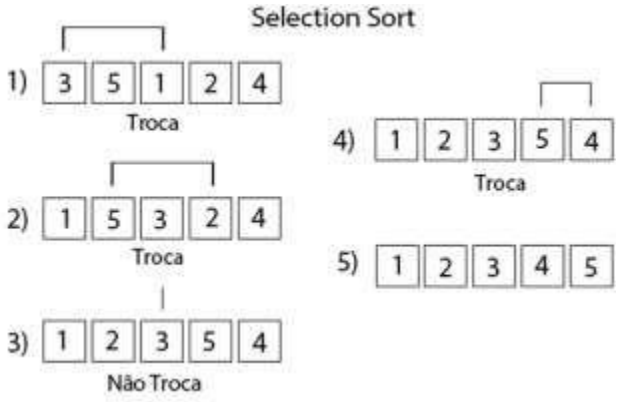

Figura 7 - Exemplo de Selection Sort

- Neste passo o primeiro número escolhido foi o 3, ele foi comparado com todos os números à sua direita e o menor número encontrado foi o 1, então **os dois trocam de lugar**.
- O mesmo processo do passo 1 acontece, o número escolhido foi o 5 e o menor número encontrado foi o 2.
- Não foi encontrado nenhum número menor que 3, então **ele fica na mesma posição**.
- O número 5 foi escolhido novamente e o único número menor que ele à sua direita é o 4, **então eles trocam**.
- Vetor já ordenado.

(Bruno – Algoritmos de ordenação).

5 Relatório com as linhas de código.

5.1 Códigos responsáveis pela importação de algumas funções.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define TAM 100
```

Figura 8 - Códigos responsáveis pela importação de algumas funções para a IDE.

Nessa estrutura de códigos pode-se observar:

- **#include <stdio.h>:** importa
- **#include <stdlib.h>:** importa
- **#include <time.h>:** importa
- **#include <math.h>:** importa
- **#define TAM 100:** define

5.2 Funções responsáveis pelas ordenações:

5.2.1 Buble Sort

```
void BubbleSort(int *v, int tam) {
    int i;

    int a, b, aux;
    for (a = tam -1; a>=1; a--) {
        for (b=0; b<a; b++) {
            if (v[b]>v[b+1]) {
                aux = v[b];
                v[b] = v[b+1];
                v[b+1] = aux;
            }
        }
    }

}
```

Figura 9 – Ordenando atraves do BubbleSort

```
void bubble(int *v,int tam)
{
    int i,j,temp,flag;
    if(tam)
        for(j=0; j<tam-1; j++)
        {
            flag=0;
            for(i=0; i<tam-1; i++)
            {
                if(v[i+1]<v[i])
                {
                    temp=v[i];
                    v[i]=v[i+1];
                    v[i+1]=temp;
                    flag=1;
                }
            }
            if(!flag)
                break;
        }
}
```

Figura 10 – Funções utilizada para o funcionamento do BubbleSort

5.2.2 Quick Sort

```
void QuickSort(int *v, int E, int D){
    int i, j, pivot, temp;

    for(i=0;i<D;i++)
        v[i] = v[i];

    if(E < D){
        pivot=E;
        i=E;
        j=D;

    while(i<j){
        while(v[i]<=v[pivot]&&i<D)
            i++;
        while(v[j]>v[pivot])
            j--;
        if(i<j){
            temp=v[i];
            v[i]=v[j];
            v[j]=temp;
        }
    }

    temp=v[pivot];
    v[pivot]=v[j];
    v[j]=temp;
    QuickSort(v, E, j-1);
    QuickSort(v, j+1,D);

    }
}
```

Figura 11 - Ordenando atraves do QuickSort

5.2.3 Insertion Sort

```
void InsertionSort(int *v, int tam){

    int i, j, tmp;

    for(i = 1; i < tam; i++){
     tmp = v[i];
     for(j = i-1; j >= 0 && tmp < v[j]; j--){
      v[j+1] = v[j];
     }
     v[j+1] = tmp;
    }
}
```

Figura 12 - Ordenando atraves do InsertionSort

5.2.4 Selection Sort

```
void SelectionSort(int *v, int tam){
    int vMenor, vAux, vTemp, vTroca;

    for(vAux = 0; vAux < tam-1; vAux++){
        vMenor = vAux;
        for(vTemp = vAux + 1; vTemp < tam; vTemp++){
            if(v[vTemp] < v[vMenor]){
                vMenor = vTemp;
            }

            if(vMenor != vAux){
                vTroca = v[vAux];
                v[vAux] = v[vMenor];
                v[vMenor] = vTroca;
            }
        }
    }
}
```

Figura 13 - Ordenando atraves do SelectionSort

5.2.5 Binary Insertion Sort

```
void BinaryInsertionSort(int *v, int tam){
    int i, loc, j, k, selected;

    for(i = 1; i < tam; i++){
        j = i - 1;
        selected = v[i];
        loc = BinarySearch(v, selected, 0, j);
        while(j >= loc){
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = selected;
    }
}
```

Figura 14 - Ordenando atraves do BinaryInsertionSort

```
int BinarySearch(int *v, int item, int low, int high){
    if(high <= low){
        return (item > v[low])? (low + 1): low;
    }

    int mid = (low + high)/2;

    if(item == v[mid]){
        return mid+1;
    }

    if(item > v[mid]){
        return BinarySearch(v, item, mid+1, high);
    }
    return BinarySearch(v, item, low, mid-1);
}
```

Figura 15 - Funções utilizada para o funcionamento do BinaryInsertionSort

5.2.6 Heap Sort

```
void HeapSort(int *vet, int n){
    int i, aux;
    for(i = (n - 1)/2; i >= 0; i--){
        criaHeap(vet, i, n-1);
    }

    for(i = n - 1; i >= 1; i--){
        aux = vet[0];
        vet[0] = vet[i];
        vet[i] = aux;
        criaHeap(vet, 0, i - 1);
    }
}
```

Figura 16 - Ordenando atraves do HeapSort

```
void criaHeap(int *vet, int i, int f){
    int aux = vet[i];
    int j = 2*i + 1;

    while(j <= f){
        if(j < f){
            if(vet[j] < vet[j + 1]){
                j = j + 1;
            }
        }

        if(aux < vet[j]){
            vet[i] = vet[j];
            i = j;
            j = 2 * i + 1;
        }else{
            j = f + 1;
        }
        vet[i] = aux;
    }
}
```

Figura 17 - Funções utilizada para o funcionamento do HeapSort

5.2.7 Merge Sort

```
void MergeSort(int *vet, int inicio, int fim){
    int meio;
    if(inicio < fim){
        meio = floor((inicio+fim)/2);
        MergeSort(vet, inicio,meio);
        MergeSort(vet, meio+1,fim);
        merge(vet, inicio,meio,fim);
    }

}
```

Figura 18 - Ordenando atraves do MergeSort

```
void merge(int *vet, int inicio, int meio, int fim){
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;
    tamanho = fim-inicio+1;
    p1 = inicio;
    p2 = meio+1;
    temp = (int *) malloc(tamanho*sizeof(int));
    if(temp != NULL){
        for(i = 0; i < tamanho; i++){
            if(!fim1 && !fim2){
                if(vet[p1] < vet[p2]){
                    temp[i] = vet[p1++];
                }else{
                    temp[i] = vet[p2++];
                }

                if(p1 > meio) fim1 = 1;
                if(p2 > fim) fim2 = 1;
            }else{
                if(!fim1){
                    temp[i] = vet[p1++];
                }else{
                    temp[i] = vet[p2++];
                }
            }
        }
        for(j = 0, k = inicio; j < tamanho; j++, k++){
            vet[k] = temp[j];
        }
    }
    free(temp);
}
```

Figura 19 - Funções utilizada para o funcionamento do MergeSort

5.2.8 Bucket Sort

```
void BucketSort(int *v, int n){
    int i,j,maior,menor,nbaldes,pos;
    struct balde *bd;
    maior=menor=v[0];

    for(i=1; i<n; i++)
    {
        if(v[i]>maior)
        {
            maior = v[i];
        }
        if(v[i]<menor)
        {
            menor = v[i];
        }
    }

    nbaldes = (maior-menor) / TAM + 1;

    bd = (struct balde *)malloc(nbaldes * sizeof(struct balde));

    for(i=0; i<nbaldes; i++)
    {
        bd[i].qtd=0;
    }

    for(i=0; i<n; i++)
    {
```

Figura 20 - Ordenando atraves do BucketSort

```
        pos=(v[i]-menor)/TAM;
        bd[pos].vl[bd[pos].qtd]=v[i];
        bd[pos].qtd++;
    }
    pos = 0;
    for(i=0; i<nbaldes; i++)
    {
        bubble(bd[i].vl,bd[i].qtd);
        for(j=0; j<bd[i].qtd; j++)
        {
            v[pos] = bd[i].vl[j];
            pos++;
        }
    }
    free(bd);
}
```

Figura 21 – Continuação do codigo de ordenação do BucketSort

5.3 Imprimir e salvar a lista não ordenada

```
void imprimeVetor(int *v, int tam){
    int i;
        for(i=0;i<tam;i++)
            printf("\nv[%d]: %d", i,v[i]);
}

void copiaVetor(int *v1, int *v2, int tam){
    int i;
    for(i = 0; i < tam; i++){
        v2[i] = v1[i];
    }
}
```

Figura 22 **– Funções de Imprir e armazena lista desordenada**

5.4 Insira o valor a ser ordenado

```
int main(int argc, char *argv[]){

  // ======== Declaração de Variaveis
  int i, tam;


  // ======== Entrando com o tamanho do vetor
  printf("Digite o tamanho do vetor: ");
  scanf("%d", &tam);
    int *vetor1;
    int *vetor2;
```

Figura 23 **– Inicio do codigo**

5.5 Alocando o tamanho do vetor em um ponteiro

```
// ======== Alocando o tamanho do vetor em um ponteiro
vetor1 = (int *)(malloc(tam * sizeof(int)));
vetor2 = (int *)(malloc(tam * sizeof(int)));

if( vetor1 == NULL || vetor2 == NULL)
{
  printf("\nErro de alocacao de memoria");
  system("pause");
  exit(1);
}

printf("\n");
```

Figura 24 **- Ponteiro**

5.6 Gerando valor aleatorio para o vetor

```
// ======== Gera valores aleatorios para o vetor
for( i = 0; i < tam; i++)
{
  vetor1[i] = rand() % tam;
}
copiaVetor(vetor1, vetor2, tam);
```

Figura 25 **– Gera o valor aleatorio e salva em outra velor**

5.7 Imprime o valor desordenado e chama para ordenação

```
// ======== Printa o vetor desordenado
 //imprimeVetor(vetor2, tam);
 printf("\n\n");


 // ======== Chamadas dos algoritimos de ordenação
 clock_t begin;
 clock_t end;
```

Figura 26 **– Função clock**

5.8 Ordenando e imprimindo na tela

```
//BubbleSort
begin = clock();
BubbleSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao BubbleSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 27 – **Chama a função de ordenação BubbleSort**

```
// QuickSort
begin = clock();
QuickSort(vetor2, 0, tam);
end = clock();
printf("Tempo de Execucao QuickSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 28 – **Chama a função de ordenação QuickSort**

```
// InsertionSort
begin = clock();
InsertionSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao InsertionSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 29 – **Chama a função de ordenação InsertionSort**

```
// BinaryInsertionSort
begin = clock();
BinaryInsertionSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao BinaryInsertionSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 30 – **Chama a função de ordenação BinaryInsertionSort**

```
// SelectionSort
begin = clock();
SelectionSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao SelectionSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 31 – **Chama a função de ordenação SelectionSort**

```
// HeapSort
begin = clock();
HeapSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao HeapSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 32 – **Chama a função de ordenação HeapSort**

```
// MergeSort
begin = clock();
MergeSort(vetor2, 0, tam-1);
end = clock();
printf("Tempo de Execucao MergeSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);

copiaVetor(vetor1, vetor2,tam);
```

Figura 33 – **Chama a função de ordenação MergeSort**

```
// BucketSort
begin = clock();
BucketSort(vetor2, tam);
end = clock();
printf("Tempo de Execucao BucketSort: %f seconds\n", (double)(end - begin) / CLOCKS_PER_SEC);
```

Figura 34 – **Chama a função de ordenação BucketSort**

5.9 Finalização do programa

```
    //imprimeVetor(vetor2, tam);
    printf("\n\n");

    free(vetor1);
    free(vetor2);

    system("pause");
    return 0;
}
```

Figura 35 – **Imprime duas linhas vazias e finaliza o codigo**

6 Bibliografia.

Técnicas para fazer a gestão do tempo, por Gabriel Marquez
https://nfe.io/blog/gestao-empresarial/gestao-do-tempo-no-trabalho/
Acesso em: 18/09/21

GESTÃO DE TEMPO, por Ietec
https://ietec.com.br/blog/como-fazer-uma-gestao-de-tempo-eficiente-no-ambiente-de-trabalho/
Acesso em: 18/09/21

Estrutura de Dados e Algoritmos, por Anderson Froes
https://digitalinnovation.one/artigos/aprenda-o-que-sao-estrutura-de-dados-e-algoritmos-material-curso-dio
Acesso em: 19/09/21

Endereços e ponteiros, por Paulo Feofiloff
https://www.ime.usp.br/~pf/algoritmos/aulas/pont.html
Acesso em: 19/09/21

Ponteiros em C, por Eduardo Casavella
http://linguagemc.com.br/ponteiros-em-c/
Acesso em: 19/09/21

Apontadores/ Ponteiros, por PUCRS
https://www.inf.pucrs.br/~pinho/PRGSWB/Ponteiros/ponteiros.html
Acesso em: 19/09/21

Vetores – arrays em linguagem C, por Eduardo Casavella
http://linguagemc.com.br/vetores-ou-arrays-em-linguagem-c/#:~:text=O%20vetor%20%C3%A9%20uma%20estrutura,inteiro%20denominado%20%C3%ADndice%20do%20vetor.
Acesso em: 27/09/21

Casting, por IME USP
https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/cast.html
Acesso em: 27/09/21

O que significa Casting, por Wagner Gaspar
https://wagnergaspar.com/casting-ou-conversao-de-tipos-na-linguagem-c/
Acesso em: 27/09/21

O que significa Casting, por Wagner Gaspar
https://youtu.be/FO8z0HQcRKA
Acesso em: 27/09/21

Alocação dinâmica de matrizes, por UFPR
https://www.inf.ufpr.br/roberto/ci067/14_alocmat.html
Acesso em: 01/10/21

Alocacao Dinamica de Vetor Linguagem C, por Eduardo Casavella
http://linguagemc.com.br/alocacao-dinamica-de-memoria-em-c/
Acesso em: 02/10/21

Alocacao Dinamica de Vetor Linguagem C, por Eduardo Casavella
https://youtu.be/avfxX-Bwjh8
Acesso em: 02/10/21

Medir tempo de execução em C, por Wurthmann
http://wurthmann.blogspot.com/2015/04/medir-tempo-de-execucao-em-c.html
Acesso em: 07/10/21

Algoritmos de Ordenação, por Domingos Lacerda Monteiro
https://silo.tips/download/algoritmos-de-ordenaao
Acesso em: 10/10/21

Bucket Sort, por GeeksforGeeks

https://www.geeksforgeeks.org/bucket-sort-2/
Acesso em: 10/10/21

Binary Insertion Sort, por Interview Kickstart
https://www.interviewkickstart.com/learn/binary-insertion-sort
Acesso em: 12/10/21

Linguagem C - Tipos Básicos, por Fabio dos Reis
https://youtu.be/n68tJh2mIx4
Acesso em: 11/10/21

Linguagem C - Declaração e Atribuição de Variáveis, por Fabio dos Reis
https://youtu.be/WC-HDwkMgGA
Acesso em: 11/10/21

Linguagem C - Função printf, por Fabio dos Reis
https://youtu.be/Ggpc4AMvDrI
Acesso em: 13/10/21

Linguagem C - Função printf, por Fabio dos Reis
https://youtu.be/MqS8vGHlTIs
Acesso em: 13/10/21

Linguagem C - Função printf, por Fabio dos Reis
https://youtu.be/2i-h3QkNJww
Acesso em: 13/10/21

Linguagem C - Função printf, por Fabio dos Reis
https://youtu.be/ijobHLHA8CU
Acesso em: 13/10/21

Linguagem C - Função scanf, por Fabio dos Reis
https://youtu.be/WoRvdhw6Bq0
Acesso em: 14/10/21

Linguagem C - Operadores e Expressões Aritméticas, por Fabio dos Reis
https://youtu.be/VZYAUQE8QBs
Acesso em: 14/10/21

Linguagem C - Desvio Condicional Composto, por Fabio dos Reis
https://youtu.be/Q27XvQFmkyQ
Acesso em: 15/10/21

Linguagem C - Desvio Condicional Aninhado, por Fabio dos Reis
https://youtu.be/7ZL8tHLTTfs
Acesso em: 15/10/21

Linguagem C - Estrutura de Repetição While, por Fabio dos Reis
https://youtu.be/zZ98f-wMirc
Acesso em: 16/10/21

Linguagem C - Estrutura de Repetição For, por Fabio dos Reis
https://youtu.be/qnrk51JoLjQ
Acesso em: 16/10/21

Linguagem C - Funções, por Fabio dos Reis
https://youtu.be/tzBq7_Cn_D4
Acesso em: 16/10/21

Linguagem C - Escopo das Variáveis, por Fabio dos Reis
https://youtu.be/GGkaN_LwR0w
Acesso em: 16/10/21

Linguagem C - Arrays, por Fabio dos Reis
https://youtu.be/nD88UCeOLKk
Acesso em: 17/10/21

Linguagem C - Ordenando Arrays, por Fabio dos Reis
https://youtu.be/pbijN3GLigM
Acesso em: 17/10/21

Linguagem C - Matrizes, por Fabio dos Reis
https://youtu.be/lEW94P355Qs
Acesso em: 17/10/21

Linguagem C - Ponteiros, por Fabio dos Reis
https://youtu.be/nC9myRXi65s
Acesso em: 18/10/21

Linguagem C - Ponteiros, por Fabio dos Reis
https://youtu.be/PgkrW9n7YiM
Acesso em: 18/10/21

Linguagem C - Estruturas, por Fabio dos Reis
https://youtu.be/oCko0q_gi_o
Acesso em: 18/10/21