Logotipo, nome da empresa

Descrição gerada automaticamente

Curso: Ciências da Computação

Campus: Vargas – Ribeirão Preto

77B1 – ATIVIDADES PRÁTICAS SUPERVISIONADAS

Nome: Eduardo dos Santos Rodrigues RA: N604794

João Paulo Rodrigues da Cruz RA: F30IBA8

Victor Assunção Silva RA: N614EB3

Outubro, 2021

Índice

Objetivo e Motivação do Trabalho...................................................03

Introdução........................................................................................04

Regras e Funcionamento do Software............................................06

Plano de Desenvolvimento do Software..........................................14

Projeto.............................................................................................21

Relatório com as Linhas de Código.................................................24

Bibliografia.......................................................................................39

Objetivo e Motivação do Trabalho

O nosso projeto de atividades práticas supervisionadas (APS) tem como objetivo desenvolver algoritmos de ordenação, comparar os tempos de execução utilizando tamanho de entradas diferentes e logo após fazer um relatório com os resultados. Com isso vamos ter parâmetros e dimensões para poder julgar e classificar métodos de entradas e suas compilações, com isso podemos definir qual a melhor forma de ordenação para determinada ocasião.

A motivação desse trabalho além de fins acadêmicos consiste em nos dar um conhecimento aprofundado no quesito de métodos de ordenação, nos mostrando que mesmo alguns deles sendo muito parecidos, no fim das contas possuem uma diferença que nos dá o lucro de milésimos de segundos. Isso em algumas áreas é preciosíssimo, nos poupando tempo de execução e tornando o ambiente de trabalho em teoria mais produtivo.

Mão segurando aparelho eletrônico

Descrição gerada automaticamente

Esses métodos estão presentes no nosso cotidiano sem ao menos percebemos, como exemplo o ato de ordenar cartas que se formos pesquisar a fundo nos leva ao método de ordenação por inserção, um dos métodos que será apresentado e demonstrado no projeto abaixo.

Introdução

Este documento tem por objetivo apresentar a construção de um programa que contêm algoritmos de ordenação e comparar os seus tempos de execução para tamanhos de entradas diferentes e fazer um relatório com os resultados e conclusões. Mostrando desde seu código principal e detalhando todas as linhas e métodos, até sua parte interativa exibindo o relatório e detalhando a utilização de todos os métodos de ordenação.

A ordenação tem como objetivo facilitar as pesquisas e buscas de determinado elemento em um conjunto ordenado, imagine como seria buscar um número em uma lista telefônica anual que possuem o telefone de milhares de pessoas e empresas sem uma ordenação alfabética, quase impossível encontrar o número que está em mente.

Jornal com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

Levando para um exemplo mais atual do que lista telefônica, imagine pesquisar algo no Google sem termos a ordenação por tópicos e/ou assuntos, todos os itens de pesquisas estariam jogados em uma página e você teria que passar de um por um para achar o desejado, isso sim seria impossível pensando na imensidão do Google. Ainda bem que a ordenação responsiva, detalhada e

confortável do Google nos proporciona uma navegação sugestiva e aconchegante.

Interface gráfica do usuário

Descrição gerada automaticamente

A computação possui uma serie de algoritmos com diversas técnicas de ordenação para organizar um conglomerado de dados, sendo conhecidos como “Métodos de Ordenação” ou “Algoritmos de Ordenação”.

Esses métodos de ordenação se classificam em 2 subgrupos, sendo eles:

- Ordenação Interna: onde todos os elementos conseguem ser armazenados na memória principal e qualquer registro tem acesso imediato.

-Ordenação Externa: onde os elementos não conseguem ser armazenados na memória principal e os registros são acessados em grandes blocos ou sequencialmente.

Esse projeto consistira em uma aplicação C que irá apresentar os seguintes algoritmos de ordenação: Ordenação por troca, BubbleSort (método da bolha), QuickSort (método da troca e partição), InsertionSort (método da inserção direta), BinaryInsertionSort (método da inserção direta binária), SelectionSort (método da seleção direta), HeapSort (método da seleção em árvore), MergeSort (método da intercalação) e o BucketSort (método da distribuição de chave). Sendo todos eles métodos de ordenação interna.

Regras e Funcionamento do Software

A ordenação de dados da importância em organizar dados de forma crescente ou decrescente para tornar mais fácil a pesquisa de um elemento. Especificamente, dada uma lista de *x* itens ( *r[ 0 ], r[ 1 ], r[ 2 ],...r[ n-1]* ) cada item nessa lista será nomeado de **registro**. Uma **chave** ( *k[ i ]* ) é associada a cada registro ( *r[ i ]* ). Declara-se que uma lista está **ordenada pela chave** se ( *i* ) preceder ( *j* ) e implicar que ( *k[ i ] < k [ j ]* se for crescenteou *k[ i ] > k [ j ]* se for decrescente ) em alguma ordenação nas chaves. No caso da computação essa organização tem um custo computacional em tempo de processamento e memória.

Nosso objetivo nesse projeto foi encontrar um algoritmo de ordenação que use pouca memória e que processe o mais rápido possível, para isso foi feito vários testes e análises de diversos métodos de ordenação, vamos apresentar e descrever cada método utilizado abaixo:

* **Ordenação por Troca**

É um método de classificação que consiste na comparação de pares de chaves de ordenação, trocando os elementos correspondentes se estiverem fora de ordem, dentro dessa classificação existem diversos métodos, como:

**- Método da Bolha (BubbleSort):**

Nesse método o princípio geral é simples, acontece trocas em todos os pares consecutivos de chaves não ordenados, e quando não restar mas pares não ordenados o vetor estará ordenado. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Em cada passo, o elemento é comparado com seu sucessor
2. Se o elemento estiver fora de ordem a troca acontece.
3. Após realizar os passos anteriores diversas vezes chegara um momento em que não será necessário ocorrer mais trocas, nesse caso a lista está ordenada.

Exemplo: 25, 57, 48, 37, 12, 92, 86, 33

Primeira etapa:

x[0] com x[1] (25 com 57) Nenhuma troca realizada

x[1] com x[2] (57 com 48) Troca realizada

x[2] com x[3] (57 com 37) Troca realizada

x[3] com x[4] (57 com 12) Troca realizada

x[4] com x[5] (57 com 92) Nenhuma troca realizada

x[5] com x[6] (92 com 86) Troca realizada

x[6] com x[7] (92 com 33) Troca realizada

Resultado: 25, 48, 37, 12, 57, 86, 33, 92

Observe que depois da primeira etapa, o maior elemento está em sua posição correta. Então o processo é realizado diversas vezes encontrando a posição correta dos números em forma decrescente.

1ª Etapa: 25, 48, 37, 12, 57, 86, 33, 92;

2ª Etapa: 25, 37, 12, 48, 57, 33, 86, 92;

3ª Etapa: 25, 12, 37, 48, 33, 57, 86, 92;

4ª Etapa: 12, 25, 37, 33, 48, 57, 86, 92;

5ª Etapa: 12, 25, 33, 37, 48, 57, 86, 92; (Aqui já está ordenado, porém o método ainda não terminou de comparar todos os números)

6ª Etapa: 12, 25, 33, 37, 48, 57, 86, 92;

7ª Etapa: 12, 25, 33, 37, 48, 57, 86, 92;

**- Método da troca e partição (QuickSort):**

A ordenação QuickSort é o algoritmo de ordenação mais utilizado por ser conhecido como o método mais eficiente. Esse método trabalha particionando um vetor em duas partes e logo após ordena-o separadamente. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Escolha um elemento da lista (pivô) para ser colocado em sua posição correta.
2. Divida a lista em duas matrizes, colocando à sua esquerda todos os valores menores que ele, e a sua direita todos os maiores (valores iguais ao pivô podem ficar de qualquer lado). Está etapa é nomeada de partição;
3. Repita a etapa acima até que as submatrizes estejam ordenadas, ocasionando a ordenação da matriz.

Exemplo: 25, 57, 48, 37, 12, 92, 86, 33

Vamos supor que o pivô escolhido é o 25 para ser colocado na posição correta, então a lista ficara:

Esquerda = (12), [ 25 ], Direita = (57, 48, 37, 92, 86, 33)

Automaticamente a submatriz (12) já foi classificado, sendo que ele é o único elemento a esquerda do pivô, então repetiremos o processo escolhendo um novo pivô para a submatriz da direita.

2ª Etapa: 12, 25, Esquerda = (48,37,33), [ 57 ], (92, 86) – Pivô = 57;

3ª Etapa: 12, 25, Esquerda = (37, 33), [ 48 ], 57, Direita = (92, 86) – Pivô = 48;

4ª Etapa: 12, 25, Esquerda = (33), [ 37], 48, 57, Direita = (92, 86) – Pivô = 37;

5ª Etapa: 12, 25, [ 33 ], 37, 48, 57, Direita = (92, 86) – Pivô = 33;

6ª Etapa: 12, 25, 33, 37, 48, 57, Esquerda = (86), [ 92 ] – Pivô = 92;

7ª Etapa: 12, 25, 33, 37, 48, 57, 86, 92 – Lista Ordenada.

* **Ordenação por Inserção**

Este método consiste em realizar a ordenação pela inserção de cada um dos elementos em sua devida posição correta, levando em consideração os demais elementos que já foram ordenados. Um exemplo dessa ordenação no cotidiano é o ato de organizar cartas de baralho.

**- Método da inserção direta (InsertionSort):**

A ordenação InsertionSort consiste em dividir um vetor em dois subvetores, o primeiro terá os elementos já ordenados e o segundo os elementos que estão em processo de ordenação. Inicialmente o primeiro subvetor terá apenas o primeiro elemento da lista e o segundo subvetor terá os demais elementos. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. É retirado o primeiro elemento do vetor não ordenado e colocado no subvetor ordenado na sua posição correta;
2. O processo acima é repetido, porém quando um segundo elemento é inserido no subvetor dos ordenados ele é comparado com os demais elementos e colocado em sua posição correta.

Exemplo: 27, 12, 20, 37, 19, 17, 15

Subvetor ordenado = O;

Subvetor não ordenado = N;

1ª Etapa: O: 27 | N: 12, 20, 37, 19, 17, 15;

2ª Etapa: O: 12, 27 | N: 20, 37, 19, 17, 15;

3ª Etapa: O: 12, 20, 27 | N: 37, 19, 17, 15;

4ª Etapa: O: 12, 20, 27, 37 | N: 19, 17, 15;

5ª Etapa: O: 12, 19, 20, 27, 37 | N: 17, 15;

6ª Etapa: O: 12, 17, 19, 20, 27, 37 | N: 15;

7ª Etapa: O: 12, 15, 17, 19, 20, 27, 37 - Vetor ordenado.

**- Método da inserção direta binária (BinaryInsertionSort):**

A ordenação BinaryInsertionSort é semelhante a ordenação acima mas com um toque especial, ao invés de retirar o primeiro elemento e descobrir sua posição no subvetor ordenado a pesquisa binária funciona localizando o meio da matriz para localizar os elementos. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Assumimos que o primeiro elemento já está ordenado e movemos para o subvetor ordenado;
2. Pegamos o segundo elemento e o armazenamos a esquerda do primeiro elemento do subvetor ordenado;
3. Se ele for menor que o número a direita ir para o próximo elemento, caso contrário inverter com o número a direita;
4. Repetir os passos acima até que a lista esteja ordenada.

Exemplo: 27, 12, 20, 37, 19, 17, 15;

Subvetor ordenado = O;

Subvetor não ordenado = N;

1ª Etapa: O: 27 | N: 12, 20, 37, 19, 17, 15;

2ª Etapa: O: 12, 27 | N: 20, 37, 19, 17, 15;

3ª Etapa: O: 20, 12, 27 | N: 37, 19, 17, 15;

4ª Etapa: O: 12, 20, 27 | N: 37, 19, 17, 15;

5ª Etapa: O: 37, 12, 20, 27 | N: 19, 17, 15;

6ª Etapa: O: 12, 20, 27, 37 | N: 19, 17, 15;

7ª Etapa: O: 19, 12, 20, 27, 37 | N: 17, 15;

8ª Etapa: O: 12, 19, 20, 27, 37 | N: 17, 15;

9ª Etapa: O: 17, 12, 19, 20, 27, 37 | N: 15;

10ª Etapa: O: 12, 17, 19, 20, 27, 37 | N: 15;

11ª Etapa: O: 15, 12, 17, 19, 20, 27, 37;

12ª Etapa: O: 12, 15, 17, 19, 20, 27, 37.

* **Ordenação por Seleção**

Este processo de ordenação consiste em uma seleção sucessiva do menor ou do maior valor contido na lista, isso irá depender se será uma lista de ordem crescente ou decrescente.

**- Método da seleção direta (SelectionSort):**

A ordenação SelectionSort classificara uma lista encontrando repetidamente o elemento superior ou inferior (se for crescente ou decrescente) e colocando esse elemento no início ou final, sendo assim essa ordenação terá dois subvetores, o que já está ordenado e o que não está ordenado. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. É feita uma observação que contém os elementos não ordenados, identificando o elemento de menor (ou maior caso seja decrescente) valor;
2. O elemento identificado acima é inserido no final do subvetor dos ordenados;
3. O tamanho do subvetor dos não ordenados vai ser diminuído em 1 a cada etapa;
4. O processo é repetido diversas vezes até que fique sobrando apenas o maior (ou menor em caso de ordem decrescente) valor da lista.

Exemplo: 21, 27, 12, 20, 37, 19, 17, 15

Subvetor ordenado = O;

Subvetor não ordenado = N;

1ª Etapa: O: 12 | N: 27, 21, 20, 37, 19, 17, (15); Vetor N = 7

2ª Etapa: O: 12, 15 | N: 21, 20, 37, 19, (17), 27; Vetor N = 6

3ª Etapa: O: 12, 15, 17 | N: 20, 37, (19), 21, 27 Vetor N = 5

4ª Etapa: O: 12, 15, 17, 19 | N: 37, (20), 21, 27 Vetor N = 4

5ª Etapa: O: 12, 15, 17, 19, 20 | N: 37, (21), 27; Vetor N = 3

6ª Etapa: O: 12, 15, 17, 19, 20, 21 | N: 37, (27); Vetor N = 2

7ª Etapa: O: 12, 15, 17, 19, 20, 21, 27 | N: (37); Vetor N = 1

8ª Etapa: O: 12, 15, 17, 19, 20, 21, 27, 37; - Vetor Ordenado

**- Método da seleção em árvore (HeapSort):**

A ordenação HeapSort é um método de ordenação em que o princípio de funcionamento é semelhando a ordenação SelectionSort (Método da Seleção Direta). Seleciona-se o menor (ou maior elemento se for decrescente), move-o para um subvetor e compara para ver se é menor ou maior. O que difere do método de ordenação acima é uma estrutura de dados chamada de fila de prioridades, dando prioridade para os elementos de maior ou menor valor. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Determine a lista;
2. O maior item da lista é armazenado na raiz da arvore. Substitua-o pelo último item da arvore, assim reduzindo o tamanho da arvore pois se tornara a raiz dela;
3. Repita o processo acima várias vezes até que o tamanho da arvore não seja maior que.

**- Método da intercalação (MergeSort):**

Assim como o QuickSort, o MergeSort é um algoritmo que particiona a lista para resolvê-la, assemelhando com o QuickSort na MergeSort divide-se a lista em 2 subvetores de tamanhos iguais, e então ele mescla pares de subvetores a ordena-os até no fim sobrar apenas a lista ordenada. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Divida a lista em n subvetores de tamanhos iguais;
2. Intercale pares adjacentes (no caso de vetores de tamanho dois);
3. Repita o processo acima até que tenhamos a lista ordenada.

Exemplo: 25, 57, 38, 37, 12, 92, 86, 33.

1ª Etapa: [25, 57] [37, 38] [12, 92] [33, 86];

2ª Etapa: [25, 37, 38, 57] [12, 33, 86, 92];

3ª Etapa: [12, 25, 33, 37, 38, 57, 86, 92]. – Vetor ordenado

**- Método da distribuição de chave (BucketSort):**

A ordenação BucketSort ou BinSort é um algoritmo de ordenação que divide o vetor em um número finito de recipientes. Logo então os recipientes são ordenados individualmente, podendo ser ordenados por algum outro algoritmo ou pelo próprio BucketSort. Após a ordenação os baldes são concatenados gerando a lista totalmente ordenada. Para exemplificar veja abaixo o passo-a-passo e algumas representações:

1. Criar N listas ligadas para dividir o intervalo em subintervalos de tamanho homogêneo.
2. Adiciona cada elemento em seu balde apropriado.
3. Unir os baldes.

Exemplo: 25, 14, 31, 53, 48, 8, 51, 43, 89, 15.

Terão 10 baldes cada um suportando números de 0..10, 11..20, 21..30....91..100.

Interface gráfica do usuário, Aplicativo, Tabela

Descrição gerada automaticamente

Plano de desenvolvimento do software

Nesse ponto do projeto iremos detalhar e apresentar a funcionalidade do nosso projeto, a estrutura do mesmo foi dívida em 3 (três) partes no mesmo arquivo, a estrutura dos algoritmos de ordenação, a Main e as saídas de cada algoritmo de ordenação. Abaixo vemos a estrutura do nosso projeto mais detalhado.

* Estrutura dos algoritmos de ordenação

Nessa parte do arquivo contém os códigos de funcionamento de cada método de ordenação, praticamente todo o conceito visto nas páginas anteriores em formato de código. Ele é apenas o conceito, nesta parte do código ainda não é possível inserir elementos e ordená-los.

Texto

Descrição gerada automaticamente

Figura 10 – Método BubbleSort

Texto

Descrição gerada automaticamente com confiança média

Figura 11 – Método QuickSort

Texto

Descrição gerada automaticamente com confiança média

Figura 12 – Método InsertionSort

Texto

Descrição gerada automaticamente

Figura 13 – Método BinaryInsertionSort

Texto

Descrição gerada automaticamente

Figura 14 – Método SelectionSort

Texto

Descrição gerada automaticamente

Figura 15 – Método HeapSort

Texto

Descrição gerada automaticamente com confiança baixa

Figura 15 – Método MergeSort

Uma imagem contendo Texto

Descrição gerada automaticamente

Figura 16.2 – Método MergeSort

* Main (Método ImprimeVetor)

Nessa parte do arquivo consta a Main do nosso projeto ou seja aqui é onde a mágica acontece, todos os nossos métodos de ordenação serão executados nesse ponto do projeto e logo após irão ser cronometrados para podermos comparar seus resultados. Abaixo veja a estrutura de funcionamento desse método.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Figura 17 – Método ImprimeVetor

* Saída do algoritmo

Essa é a parte final do nosso projeto, aqui é onde o programa nos apresenta o resultado da parte anterior (Método ImprimeVetor), desse jeito deixamos o projeto um pouco mais organizado visualmente.

Texto

Descrição gerada automaticamente

Figura 120 – Saída do tempo de execução dos métodos de ordenação

Interface gráfica do usuário

Descrição gerada automaticamente com confiança baixa

Figura 120.2 – Saída do tempo de execução dos métodos de ordenação

Interface gráfica do usuário, Site

Descrição gerada automaticamente

Figura 120.3 – Saída do tempo de execução dos métodos de ordenação

Projeto

Neste ponto do projeto iremos apresentar os resultados finais mostrando o compilamento do código juntamente com um relatório gerado no Excel de seus tempos de execução cronometrados pelo próprio sistema. Com isso poderemos analisar os dados obtidos e debater sobre os mesmos.

Texto

Descrição gerada automaticamente

Como já foi citado anteriormente o método de ordenação de algoritmos QuickSort (Método da troca e partição) é um dos mais utilizados por conta de dois fatores, seu tempo de execução é mais baixo em relação com outros métodos e não menos importante seu consumo de memória e bem baixo em relação com os outros métodos de ordenação de algoritmos.

Tabela

Descrição gerada automaticamente

A quantidade de consumo de alocação de memória de cada método de ordenação de algoritmo não foi possível medir por conta da falta de conhecimento do mesmo. Porém como já afirmado acima o método de ordenação QuickSort (Método da troca e partição) segundo fontes em fóruns e bibliotecas de desenvolvedores da linguagem é considerado um dos melhores em consumo de alocação de memória.

Uma imagem contendo Linha do tempo

Descrição gerada automaticamente

**Link do Projeto no GitHub:** https://github.com/duzinho/APS---ALGOTIMO-DE-ORDENA-O.git

Relatório com as Linhas de Código do Programa

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

// ----- BUBBLE 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;

}

}

}

}

// ----- 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);

}

}

// ----- INSERTION SORT -----

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

int i, j, tmp;

for(i=0;i<tam;i++){

printf("\n v[%d]:", i);

//scanf("%i", &v[i]);

}

for(i=1 ; i<5 ; i++){

tmp = v[i];

j = i - 1;

while((j>=0) && (v[j]>tmp)){

v[j+1] = v[j];

j = j - 1;

}

v[j+1] = tmp;

}

}

// ----- BINARY INSERTION SORT -----

void BinaryInsertionSort(int a[], int n)

{

int i, loc, j, k, selected;

for (i = 1; i < n; ++i)

{

j = i - 1;

selected = a[i];

loc = BinarySearch(a, selected, 0, j);

while (j >= loc)

{

a[j+1] = a[j];

j--;

}

a[j+1] = selected;

}

}

int BinarySearch(int a[], int item,

int low, int high)

{

if (high <= low)

return (item > a[low]) ?

(low + 1) : low;

int mid = (low + high) / 2;

if (item == a[mid])

return mid + 1;

if (item > a[mid])

return BinarySearch(a, item,

mid + 1, high);

return BinarySearch(a, item, low,

mid - 1);

}

// ----- 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;

}

}

}

}

// ----- HEAP SORT -----

void HeapSort(int a[], int n) {

int i = n / 2, pai, filho, t;

while(true) {

if (i > 0) {

i--;

t = a[i];

} else {

n--;

if (n <= 0) return;

t = a[n];

a[n] = a[0];

}

pai = i;

filho = i \* 2 + 1;

while (filho < n) {

if ((filho + 1 < n) && (a[filho + 1] > a[filho]))

filho++;

if (a[filho] > t) {

a[pai] = a[filho];

pai = filho;

filho = pai \* 2 + 1;

} else {

break;

}

}

a[pai] = t;

}

}

// ----- MERGE SORT -----

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

void MergeSort(int arr[], int l, int r)

{

if (l < r)

{

int m = l+(r-l)/2;

MergeSort(arr, l, m);

MergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

void printArray(int A[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

// ----- BUCKET SORT -----

struct bucket

{

int count;

int\* value;

};

int compareIntegers(const void\* first, const void\* second)

{

int x = \*((int\*)first), y = \*((int\*)second);

if (x == y)

{

return 0;

}

else if (x < y)

{

return -1;

}

else

{

return 1;

}

}

void BucketSort(int array[],int n)

{

struct bucket buckets[3];

int i, j, k;

for (i = 0; i < 3; i++)

{

buckets[i].count = 0;

buckets[i].value = (int\*)malloc(sizeof(int) \* n);

}

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

{

if (array[i] < 0)

{

buckets[0].value[buckets[0].count++] = array[i];

}

else if (array[i] > 10)

{

buckets[2].value[buckets[2].count++] = array[i];

}

else

{

buckets[1].value[buckets[1].count++] = array[i];

}

}

for (k = 0, i = 0; i < 3; i++)

{

qsort(buckets[i].value, buckets[i].count, sizeof(int), &compareIntegers);

for (j = 0; j < buckets[i].count; j++)

{

array[k + j] = buckets[i].value[j];

}

k += buckets[i].count;

free(buckets[i].value);

}

}

void imprimeVetor(int \*v, int tam){

int i;

for(i=0;i<tam;i++)

printf("\nv[%d]: %d", i,v[i]);

}

int main(int argc, char \*argv[])

{

int i, tam;

printf("Digite o tamanho do vetor: ");

scanf("%d", &tam);

int vetor[tam];

printf("\n");

for(i = 0; i < tam; i++)

{

vetor[i] = rand() % 100;

}

clock\_t begin;

clock\_t end;

begin = clock();

BubbleSort(vetor, tam);

end = clock();

printf("Tempo da Execucao BubbleSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

// imprimeVetor(vetor, tam);

begin = clock();

InsertionSort(vetor, tam);

end = clock();

printf("Tempo da Execucao InsertionSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

QuickSort(vetor, 0, tam);

end = clock();

printf("Tempo da Execucao QuickSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

BinaryInsertionSort(vetor, tam);

end = clock();

printf("Tempo da Execucao BinaryInsertionSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

SelectionSort(vetor, tam);

end = clock();

printf("Tempo da Execucao SelectinSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

HeapSort(vetor, tam);

end = clock();

printf("Tempo da Execucao HeapSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

MergeSort(vetor, 0, tam);

end = clock();

printf("Tempo da Execucao MergeSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

begin = clock();

BucketSort(vetor, tam);

end = clock();

printf("Tempo da Execucao BucketSort: %f seconds\n", (double)(end - begin) / CLOCKS\_PER\_SEC );

free(vetor);

system("pause");

return 0;

}

Referências Bibliográficas

https://teams.microsoft.com/ -- Material entregue pelas aulas.

https://pt.wikipedia.org/ -- Utilizado para pesquisas breves.

https://www.interviewkickstart.com/learn/binary-insertion-sort --BinaryInsertionSort.

https://www.devmedia.com.br -- Biblioteca Dev.