UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

RESOLUÇÃO DO TRABALHO 02 – Ordenação e Busca

1) (2 pontos) Fazer uma pesquisa aprofundada com. Definição, algoritmo, vantagens e desvantagens, referências bibliográficas dos seguintes algoritmos de ordenação:

- ➢ Selection Sort;
- ➢ Insertion Sort;
- ➢ Bubble Sort;
- ➢ Radix Sort;
- ➢ Quick Sort;
- ➢ Merge Sort;
- ➢ Busca sequencial
- ➢ Busca Binária

Ordenação

Selection Sort

Definição: Este algoritmo é 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 lista esteja ordenada.

Algoritmo:

```
/* Método de Ordenação de Dados - Selection Sort */

//Declaração das bibliotecas;
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define o tamanho do vetor;
#define TAM 50

// Declara a função select sort;
void Select_Sort(int *v){
```

```
	int menor;
	int aux;
	int temp;
	int troca;

// Percorre todo o vetor até TAM-1, pois a última posição não precisa testar pois já estará ordenada;
	for(aux=0; aux < TAM-1; aux++){
		menor = aux; // Menor valor recebe a posição que está passando;

// Percorre o vetor da posição aux+1 até o final;
		for (temp=aux+1; temp < TAM; temp++){

// Testa se a posição que está passando é menor que o menor valor;
			if (v[temp] < v[menor]){
				menor = temp; // menor recebe a posição do menor valor;
			}
		}

// Se a posição for diferente da que está passando, ocorre a troca;
		if (menor != aux){
			troca = v[aux];
			v[aux] = v[menor];
			v[menor] = troca;
		}
	}
}

int main(){
	int vetor[TAM]; // Declara o vetor
	int Aux;

	srand ( time(NULL) );
	for (Aux=0; Aux < TAM; Aux++){
// Preenche o vetor aleatoriamente;
		vetor[Aux] = (rand() % 89) + 10;
		printf(" %d,",vetor[Aux]);
	}

	Select_Sort(vetor); // Chama a função de Ordenação;
	printf("\n\n");

	for (Aux=0; Aux < TAM; Aux++){
		printf(" %d,",vetor[Aux]);
	}

	printf("\n\n ");
```

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

Vantagens: Se mostra mais eficiente em listas em ordem decrescente em relação ao tempo e quantidade de movimentações.

Desvantagens: O algoritmo sempre tem que percorrer o vetor inteiro para descobrir qual é o menor valor dentre os que ainda não estão ordenados. Assim, não possui um desempenho muito bom em vetores ordenados em ordem crescente e desordenados.

Insertion Sort

Definição: O Insertion sort é um 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.

Algoritmo:

```
/*    Ordenação Insertion Sort em C    */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TAM 100 // Declara o tamanho do vetor;

int main(){
	int vetor[TAM];
	int aux;

	srand(time(NULL));
	for (aux=0; aux < TAM; aux++){
		// Preenche o vetor com valores aleatórios de 10 até 99;
		vetor[aux] = (rand() % 90) + 10;
		printf(" %d,",vetor[vAux]);
	}

	Insertion_Sort(vetor); // Chama a função de Ordenação;
	printf("\n\n");
```

```
	for (aux=0; aux < TAM; aux++){
		printf(" %d,",vetor[aux]);
	}

	printf("\n\n ");
	system("pause");
}

void Insertion_Sort(int *v){
	int aux;
	int temp;
	int troca;

	// aux começa na posição 1 do vetor e vai até a última posição;
	for (aux=1; aux < TAM; aux++){
		// temp recebe a posição que está passando no "for";
		temp = aux;

		// Enquanto o valor que está passando na posição "temp" for menor
	que a posição "temp" menos 1, ocorre a troca;
		while(vetor[temp] < vetor[temp-1]){
			troca       = vetor[temp];
			vetor[temp]   = vetor[temp-1];
			vetor[temp-1] = troca;
			temp--; // temp decrementa 1;
		}
// Quando "temp" chegar na posição 0, primeira posição do vetor, o laço while
para;
		if (temp == 0)
			break;
	}
}
```

Vantagens: Implementação implementação simples, é eficiente para pequenos conjuntos de dados, é mais eficiente na prática do que outros algoritmos como selection sort e bubble sort.

Desvantagens: É menos eficiente em grandes listas que outros algoritmos como, por exemplo o quicksort, heapsort ou merge sort.

Bubble Sort

Definição: o método Bolha consiste em percorrer todo o vetor testando a posição que está passando com a próxima posição, se o valor que está passando é maior que o próximo, ele faz a troca, quando chegar ao final do vetor, o maior número do

vetor estará na última posição, ou seja, a próxima vez que o vetor for percorrido poderemos testar uma posição a menos no vetor.

Algoritmo:

```
// Exemplo do Método Bolha de Ordenação de Dados
// Inclusão das bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TAM 100 // Define o tamanho do vetor

int main(){
	int vetor[TAM]; // Declara o vetor de valores inteiros
	int aux;

	srand ( time(NULL) );
	printf("\n\t Dados não Ordenados\n\n");
	for ( aux=0; aux<TAM; aux++ ){
		// Preenche o vetor randomicamente com valores de 10 a 99
		vetor[aux] = (rand() % 89) + 10;
		printf(" %d,",vetor[vAux]);
	}

	Bolha(vetor); // Chama a função bolha

	printf("\n\n ");
	printf("\n\t Dados Ordenados\n\n");
	for ( aux=0; aux<TAM; aux++ ){
		printf(" %d,",vetor[aux]);
	}

	printf("\n\n ");
	system("pause");
}



// Define a função Bolha
void Bolha(int *v){
	int aux;
	int temp;
	int troca;

	// Percorre o vetor de trás para a frente
	for( aux = TAM-1; aux >= 0;aux-- ){
		// Percorre o vetor de frente para trás, até a posição que está
passando no primeiro "for", ou seja, cada vez que passar pelo primeiro "for"
ele vai testar uma posição a menos aqui.
		for( temp = 0; temp < aux; temp++ ){
		// Testa se a posição que está passando é maior que a próxima, se
for ocorre a troca
```

```
                    if( v[temp] > v[temp+1] ){
                          // Manda o valor atual para uma variável aux
                          troca = v[temp];
                          // Manda o próximo valor para a posição atual
                          v[temp] = v[temp+1];
                          // Manda a variável aux, com o valor da posição
atual, para a próxima posição
                          v[temp+1] = troca;
                    }
              }
       }
}
```

Vantagens: Bubble sort é o algoritmo mais simples.

Desvantagens: É um dos menos eficientes.

Radix Sort

Definição: O algoritmo segue o seguinte padrão:

- Receba uma matriz não ordenada de números inteiros, muitas vezes se refere / representa uma chave.
- Iterate de mais para o menos (ou menos para a maioria) dígitos significativos.
- Cada iteração classifica todos os valores do dígito significativo atual da matriz.

* Uma chave neste caso, é um valor inteiro, que pode ser associado a outros dados, como data de nascimento, localização, etc.

Algoritmo:

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vector>
```

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

```
using namespace std;

class Radix{
   private:
      int *vetor;
      int n;
   public:
      Radix(int size){
         this->vetor = new int[size];
         this->n = size;
         memset(vetor, 0, size*sizeof(vetor));
      }

      Radix(int *origem, int size){
         this->n = size;
         vetor = origem;
      }

      int *getVetor(){
         return(this->vetor);
      }

      int getVetorSize(){
         return(this->n);
      }

      void ordenacaoRadix(){
         vector<int> pilha[10];
         int iCont = 0;
         int max = getDigInteiros(getMaxValue());
         int alg = 1;
         do {
            for(int i = 0; i < 10 && !iCont; i++)
               pilha[i].clear();
            pilha[getAlgRangeInt(vetor[iCont], alg)].push_back(vetor[iCont]);
            iCont++;
            if(iCont >= this->n){
```

```
                iCont = 0;
                for(int i = 0; i < 10; i++){
                   for(int j = 0; j < pilha[i].size(); j++){
                      vetor[iCont++] = pilha[i][j];
                   }
                }
                iCont = 0;
                alg++;
             }
          } while(pilha[0].size() != this->n || (max+1) >= alg);
       }

    private:
       int getMaxValue(){
          int maior = 0;
          for(int i = 1; i < this->n; i++){
             if(*(vetor + i) > *(vetor + maior)){
                maior = i;
             }
          }
          return(*(vetor + maior));
       }

       int getAlgRangeInt(int valor, int algarismo){
          unsigned quociente, divisor;
          quociente = (int) pow(10, algarismo);
          divisor = (int) pow(10, algarismo - 1);
          return ((valor % quociente) / divisor);
       }

       int getDigInteiros(int valor){
          if(valor<10) return 1;
          return(1 + getDigInteiros(valor / 10));
       }
};

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

```
    int vetor[] = {20, 10, 2348, 22, 2, 50, 80, 5};


    Radix radix(vetor, 8);
    radix.ordenacaoRadix();


    int *retorno = radix.getVetor();


    for(int i = 0; i < 8; i++){
        cout << (i + 1) << ": " << *(retorno + i) << endl;
    }


    return(EXIT_SUCCESS);
}
```

Vantagens: Ordena vetores em tempo linear, não realiza comparações, é um algoritmo de ordenação estável.

Desvantagens: Usa dois outros vetores na ordenação, utilizando mais espaço na memória.

Quick Sort

Definição: Nele se escolhe 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.

Algoritmo:

```
/*    Quick sort    */

#include <stdio.h>
#include <stdlib.h>

// Define uma constante
#define MAX 10

void quick_sort(int *a, int left, int right);
```

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

```
// Função main
int main(){
 int i, vet[MAX];

 // Lê MAX ou 10 valores
 for(i = 0; i < MAX; i++){
      printf("Digite um valor: ");
      scanf("%d", &vet[i]);
 }

 // Ordena os valores
 quick_sort(vet, 0, MAX - 1);

 // Imprime os valores ordenados
 printf("nnValores ordenadosn");
 for(i = 0; i < MAX; i++){
      printf("%dn", vet[i]);
 }
 system("pause");
 return 0;
}

// Função de Ordenação por Seleção
// Quick sort function
void quick_sort(int *a, int left, int right) {
    int i, j, x, y;

    i = left;
    j = right;
    x = a[(left + right) / 2];

    while(i <= j) {
        while(a[i] < x && i < right) {
            i++;
        }
        while(a[j] > x && j > left) {
            j--;
        }
        if(i <= j) {
            y = a[i];
            a[i] = a[j];
            a[j] = y;
            i++;
            j--;
        }
    }

    if(j > left) {
        quick_sort(a, left, j);
    }
    if(i < right) {
        quick_sort(a, i, right);
    }
}
```

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

Vantagens: O Quicksort é o algoritmo mais eficiente na ordenação por comparação.

Desvantagens: Sua implementação é muito delicada e difícil: Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados. O método não é estável.

Merge Sort

Definição: Como estamos usando divisão e conquista para ordenar, precisamos decidir quais serão nossos subproblemas. O problema completo é ordenar um array inteiro. Vamos dizer que um subproblema é ordenar um subarray. Em particular, iremos pensar em um subproblema como ordenar o subarray começando de um índice **p** e continuamos através do índice **r**. Isso será conveniente para termos uma notação para um subarray, então vamos dizer que array[p..r] denota o subarray de array. Nós temos nossa notação, para um array de **n** elementos, podemos dizer que o problema original é ordenar o array[0..n-1].

Veja abaixo como o merge sort usa divisão e conquista:

1. Divisão pelo número encontrado qqq para a posição entre ppp e rrr. Faça isso da mesma forma que encontramos o ponto médio na busca binária: adicione ppp e rrr, divida por 2, e arredonde para baixo.
2. Conquiste organizando recursivamente os subarrays em cada dois subproblemas criados pela etapa da divisão. Isso é, organize recursivamente o subarray array[p..q] e organize recursivamente o subarray array[q+1..r].
3. Combine juntando os dois subarrays organizados de volta em um único subarray array[p..r] organizado.

Precisamos de um caso base. O caso base é um subarray contendo menos do que dois elementos, isto é, quando p \geq rp≥rp, is greater than or equal to, r, uma vez que um subarray sem elementos ou com apenas um elemento já está ordenado. Então iremos dividir, conquistar e combinar apenas quando p < rp<rp, is less than, r.

Algoritmo:

```
#include <stdio.h>
#include <stdlib.h>

void lerVet( int *p, int t ){
```

```
	int i;

	for ( i=0; i 1 ) {

		meio = t / 2;
		mergeSort(p, meio);
		mergeSort(p + meio, t - meio);
		merge(p, t);

	}
}


void main(){

	int *p, tam;

	printf("Quantidade de elementos do vetor?");
	scanf("%d",&tam);

	p = (int*) malloc(tam * sizeof(int));

	printf("\nDigite o conteudo do vetor:\n");
	lerVet(p, tam);

	printf("\nConteudo digitado para o vetor:\n");
	mostrarVet(p, tam);

	printf("\nOrdenando o vetor...\n");
	mergeSort(p, tam);

	printf("\nConteudo do vetor ja ordenado:\n");
	mostrarVet(p, tam);

	free(p);

}
```

Vantagens: É um algoritmo de ordenação de fácil implementação. Útil para Aplicações com restrição de tempo. Passível de ser transformado em estável. O Mergesort é O(n log n).

Desvantagens: Alto consumo de memória, devido à série de chamadas recursivas. Utiliza memória auxiliar – O(n).

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

Busca:

Busca sequencial

Definição: A busca sequencial é a técnica mais simples de realizar uma busca em uma lista de dados desordenados. Ela visa procurar o valor através de comparações sucessivas a partir do primeiro elemento (ou último) até que se encontre o valor desejado ou até que os elementos da estrutura se esgotem. Pode-se utilizar vetor, lista encadeada ou arquivo binário como estrutura de dados. No melhor caso, o elemento a ser buscado é encontrado logo na primeira tentativa da busca. No pior caso, o elemento a ser buscado encontra-se na última posição e são feitas N comparações, sendo N o número total de elementos. No caso médio, o elemento é encontrado após (N+1)/2 comparações. O algoritmo de busca linear é um algoritmo O(n).

Algoritmo:

```
int busca_sequencial_R(int posi, int v[MAX], int n) {
      if (posi == 0)
            return -1;
      else if (x == v[posi - 1])
            return posi - 1;
      else
            return busca_sequencial_R(posi - 1, v, n);
}


int main (){
    int vetor={7,6,4,9,2}, n,posi;

    scanf("%d", &n);

    printf("%d", busca_sequencial_R(posi, vetor, n));
}
```

Vantagens: Não precisa ser ordenada. Simplicidade de implementação.

Desvantagens: Não é tão eficiente em vetores muito grande.

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

Busca binária

Definição: A busca binária é um eficiente algoritmo para encontrar um item em uma lista ordenada de itens. Ela funciona dividindo repetidamente pela metade a porção da lista que deve conter o item, até reduzir as localizações possíveis a apenas uma. Nós usamos a busca binária em um jogo de adivinhação no tutorial introdutório.

Algoritmo:

```
#include <stdio.h>
#include <stdlib.h>
#define max 10

void le_vetor(int v[]);
int le_valor();
void ordena(int v[]);
int pesq_bin(int v[],int n, int comeco, int fim);

int main()
{     int /*v[max]*/num,p;
    struct pessoa
    {
     char nome[30];
     int telefone;
     };
    struct pessoa v[max];

       le_vetor(v);
       num=le_valor();
       ordena(v);
       p=pesq_bin(v,num,0,max-1);
       if (p>=0)
         printf("\nEncontrado, na posicao: %i do vetor",p);
       else
         printf("\nNao foi encontrado!");
    return 0;
```

```
}
void le_vetor(int v[])
{     int x;
      for (x=0;x<max;x++)
      {     printf("Digite o telefone %d: ",x+1);
            scanf("%d",&v[x]);
      }
}
int le_valor()
{ int n;
  printf("Digite o valor a procurar:");
  scanf("%d",&n);
  return n;
}
void ordena(int v[])
{
      int x,y,aux;
      for(x=0;x<max-1;x++)
       for (y=x+1;y<max;y++)
        if (v[x]>v[y])
        {  aux=v[x];
           v[x]=v[y];
           v[y]=aux;
        }
        for (x=0;x<max;x++)
            printf("Vetor ordenado: %d: \n",v[x]);
}
int pesq_bin(int v[],int n, int comeco, int fim){
      int meio,achou=-1;
      meio=(comeco+fim)/2;
      if (comeco<=fim)
    {   if (n==v[meio])
        {   achou=meio;
            return achou;
        }
        else
            if(n<v[meio])
```

```
                pesq_bin(v,n,comeco,meio-1);
            else
                pesq_bin(v,n,meio+1,fim);


    }
    else
        return achou;
}
```

Vantagens: É sem dúvida computacionalmente mais inteligente e direta na maioria das vezes.

Desvantagens: Em números pequeno, ela gasta mais tempo de processamento por causa da chamada recursiva e a ordenação que precisa ser feita antes da busca.

UNIVERSIDADE FEDERAL DE RORAIMA
PROFESSOR (A): FELIPE DWAN
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO(A): EWELLY FABIANE CUNHA DE SOUSA
DÉBORA BIANCA TAVEIRA DE MOURA

Referências bibliográficas

https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261

https://updatedcode.wordpress.com/2011/11/10/selection-sort-em-c/

https://updatedcode.wordpress.com/2011/11/14/insertion-sort-em-c/

https://updatedcode.wordpress.com/2011/11/06/bubble-sort-em-c/

https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

http://w3.ualg.pt/~hshah/ped/Aula%2014/merge_final.html

http://www.cos.ufrj.br/~rfarias/cos121/aula_07.html

https://pt.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

https://www.vivaolinux.com.br/topico/C-C++/Ordenacao-e-pesquisa-binaria-em-C

https://forum.imasters.com.br/topic/538654-busca-com-recursividade/

http://www.facom.ufms.br/~lianaduenha/sites/default/files/aula04.pdf