Nome: Rafael Ricardo M. Monteiro

Matrícula: 202202270636

Campus: Recreio

Estrutura de dados em C/C++

✓ *Métodos de Ordenação:*

- *Insertion Sort (C)*

```c
#include <stdio.h>

void printVetor(int vetor[], int tamanho) {
  for (int i = 0; i < tamanho; i++) {
    printf("%d ", vetor[i]);
  }
  printf("\n");
}
void insertionSort(int array[], int tamanho) {
  for (int step = 1; step < tamanho; step++) {
    int key = array[step];
    int j = step - 1;

    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

int main() {

  int vetor[0];
  int tamanho;
  printf("Digite o tamanho do vetor: ");
  scanf("%d", &tamanho);
  vetor[0]*tamanho;
  for(int i= 0; i < tamanho;i++ ){
    printf("Digite o %dº número do vetor:" , i+1);
    scanf("%d", &vetor[i]);
    }
  insertionSort(vetor, tamanho);
  printf("Sorteando os números em ordem crescente:\n");
  printVetor(vetor, tamanho);
}
```

- ✓ Estável: Não muda a ordem relativa de elementos com valores iguais
- ✓ É de simples implementação, leitura e manutenção;
- ✓ Muitas trocas, e menos comparações;
- ✓ requer uma quantidade constante de O (1) espaço de memória adicional
- ✓ Melhor caso: O(n), quando a matriz está ordenada.
- ✓ Médio caso: $O(n^2/4)$, quando a matriz tem valores aleatórios sem ordem de classificação (crescente ou decrescente);
- ✓ Pior caso: $O(n^2)$, quando a matriz está em ordem inversa, daquela que deseja ordenar.

- **Vantagens**:
  - É o método a ser utilizado quando o arquivo está "quase" ordenado
  - É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear.
  - O algoritmo de ordenação por inserção é estável.

- **Desvantagens**:
  - Alto custo de movimentação de elementos no vetor.

- Selection Sort (C++)

```cpp
#include <iostream>
using namespace std;

void troca(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void printVetor(int vetor[], int tamanho) {
  for (int i = 0; i < tamanho; i++) {
    cout << vetor[i] << " ";
  }
  cout << endl;
}

void selectionSort(int vetor[], int tamanho) {
  for (int step = 0; step < tamanho - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < tamanho; i++) {

      if (vetor[i] < vetor[min_idx])
        min_idx = i;
    }
    troca(&vetor[min_idx], &vetor[step]);
  }
}

int main() {
 int vetor[0];
  int tamanho;
  printf("Digite o tamanho do vetor: ");
  scanf("%d", &tamanho);
  vetor[0]*tamanho;
  for(int i= 0; i < tamanho;i++ ){
    printf("Digite o %dº número do vetor:" , i+1);
    scanf("%d", &vetor[i]);
    }
  selectionSort(vetor, tamanho);
  printf("Sorteando os números em ordem crescente:\n");
  printVetor(vetor, tamanho);
}
```

- Complexidade:
  - O selection sort compara a cada interação um elemento com os outros, visando encontrar o menor. Dessa forma, podemos entender que não existe um melhor caso mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois laços do algoritmo, o externo e o interno. A complexidade deste algoritmo será sempre $O(n^2)$ enquanto, por exemplo, os algoritmos heapsort e mergesort possuem complexidades O (n ***log*** n).

- Vantagens:
  - Ele é um algoritmo simples de ser implementado em comparação aos demais.
  - Não necessita de um vetor auxiliar (in-place).
  - Por não usar um vetor auxiliar para realizar a ordenação, ele ocupa menos memória.
  - Ele é uns dos mais velozes na ordenação de vetores de tamanhos pequenos.

- Desvantagens:
  - Ele é um dos mais lentos para vetores de tamanhos grandes.
  - Ele não é estável.

- Bubble Sort(C)

```c
#include <stdio.h>
void bubbleSort(int vetor[], int tamanho) {
  for (int step = 0; step < tamanho - 1; ++step) {
    for (int i = 0; i < tamanho - step - 1; ++i) {
      if (vetor[i] > vetor[i + 1]) {
        int temp = vetor[i];
        vetor[i] = vetor[i + 1];
        vetor[i + 1] = temp;
      }
    }
  }
}

void printVetor(int Vetor[], int tamanho) {
  for (int i = 0; i < tamanho; ++i) {
    printf("%d  ", Vetor[i]);
  }
  printf("\n");
}

int main() {
int vetor[0];
  int tamanho;
  printf("Digite o tamanho do vetor: ");
  scanf("%d", &tamanho);
  vetor[0]*tamanho;
  for(int i= 0; i < tamanho;i++ ){
    printf("Digite o %dº número do vetor:" , i+1);
    scanf("%d", &vetor[i]);
    }
  bubbleSort(vetor, tamanho);
  printf("Sorteando o vetor em ordem crescente:\n");
  printVetor(vetor, tamanho);
}
```

★ O Bubble sort, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vector diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo. No melhor caso, o algoritmo executa **N** operações relevantes, onde **n** representa o número de elementos do vector. No pior caso, são feitas $\mathbf{n^2}$ operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

- Vantagens:
  - ✓ É Estável:
  - ✓ 1 Simples de entender e implementar.
- Desvantagens:
  - Uma desvantagem é que na prática ele tem execução lenta mesmo quando comparado a outros algoritmos quadráticos (n^2);
  - Tem um número muito grande de movimentação de elementos, assim não deve ser usado se a estrutura a ser ordenada for complexa

- Merge Sort(C++)

```cpp
#include <iostream>
using namespace std;

void merge(int vetor[], int p, int q, int r) {


  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = vetor[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = vetor[q + 1 + j];


  int i, j, k;
  i = 0;
  j = 0;
  k = p;


  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      vetor[k] = L[i];
      i++;
    } else {
      vetor[k] = M[j];
      j++;
    }
    k++;
  }


  while (i < n1) {
    vetor[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    vetor[k] = M[j];
    j++;
    k++;
  }
}
```

```
48
49  void mergeSort(int vetor[], int l, int r) {
50    if (l < r) {
51
52      int m = l + (r - l) / 2;
53
54      mergeSort(vetor, l, m);
55      mergeSort(vetor, m + 1, r);
56
57      merge(vetor, l, m, r);
58    }
59  }
60
61  void printVetor(int vetor[], int tamanho) {
62    for (int i = 0; i < tamanho; i++)
63      cout << vetor[i] << " ";
64    cout << endl;
65  }
66
67  int main() {
68    int vetor[0];
69    int tamanho;
70    cout << ("Digite o tamanho do vetor: ");
71    cin >> tamanho;
72    vetor[0]*tamanho;
73    for(int i= 0; i < tamanho;i++ ){
74      cout << "Digite o " << (i+1) << "º número do vetor:";
75      cin >> vetor[i];
76      }
77    mergeSort(vetor, 0, tamanho - 1);
78
79    cout << "Sortendo o número dos vetores em ordem crescente: \n";
80    printVetor(vetor, tamanho);
81    return 0;
82  }
```

★ 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.

➢ Vantagens:

- ✓ divida o vetor em 2 sub vetores (ao meio) recursivamente até ele ter o tamanho 1.
- ✓ Intercale pares de elementos adjacentes. Repita esse processo até restar apenas um arquivo de tamanho n.
- ✓ Dividir e Conquistar;
- ✓ Divide, recursivamente, o conjunto de dados até que o subconjunto possua 1 elemento
- ✓ Combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado
- ✓ Esse processo se repete até que exista apenas 1 conjunto.
- ✓ O MergeSort é estável: não altera a ordem de dados iguais.

- ✓ Pode ser adaptado para ordenação de arquivos externos (memória secundária).

Desvantagens:

- ✓ Utiliza mais memória para poder ordenar (vetor auxiliar).