Lista Estrutura de Dados e Análise de Algoritmos
Valor: 6,00 pontos - Individual

1. Inverter a posição dos elementos de um vetor.
   a) Defina o passo base e o passo de recursão para o problema.

Estamos percorrendo os valores rumo ao centro do array, logo esse é o nosso ponto de parada, ou seja, o passo base é a metade do vetor.

Após criada a função, passamos como parâmetro o vetor *(que terá sua posição alterada)*, o início *(ou seja, zero)* e o fim *(ou seja, vetor.length - 1)*.

Dentro da condicional if, definimos que enquanto o início for menor que o fim, ocorrerá a recursão, ou seja, a função será chamada novamente dentro dela mesma. Porém, antes de chamarmos a função recursiva, precisamos primeiro:

- Pegar o elemento contido no início e salva na variável B;
- Pegar o elemento contido no fim e colocá-lo no início.
- Pegar o elemento contido no início salvo na variável B e colocá-lo no final.

Agora sim, chamamos a função novamente passando como parâmetro:
- O vetor que está sendo alterado.
- inicio + 1
  - Ou seja, o início deixa de ser 0 e se torna 1;
- fim - 1
  - Ou seja, o fim deixa de ser 9 e se torna 8;
  -

Repetimos os passos acima até que todos os elementos sejam lidos e ajustados, resumindo, assim que o caso base for atingido.

   b) Implemente um algoritmo recursivo para o problema. (github.com/chalestristian)

```
public class Q1 {
   public static void main(String[] args) {
       int[] A = {20,21,22,23,24,25,26,27,28,29,30};
       TrocarPosicao(A,0,A.length-1);
   }
  static void TrocarPosicao(int X[], int inicio, int fim){
      int B;
      if(inicio < fim){
          B = X[inicio];
          X[inicio] = X[fim];
          X[fim] = B;

          TrocarPosicao(X, inicio + 1, fim - 1);
      }
  }
}
```

Thales Cristian Eugenio - 12013012

2. Determine o que faz a função recursiva a seguir. Mostre seu raciocínio.

```
int Recursiva(int n) {
        int x;
        if n <= 0
                x = 1;
        else
                x = Recursiva(n-1) + Recursiva(n-1);
        return x;
}
```

O algoritmo tem como resultado o enésimo número referente a sequência de Fibonacci, sendo que N é igual a soma dos elementos anteriores.

Consiste na tradução direta de recorrência para o algoritmo recursivo.

3. Defina brevemente o funcionamento dos algoritmos de ordenação (BubbleSort, SelectionSort, InsertionSort, QuickSort e MergeSort) e destaque em quais situações os algoritmos são recomendados.

BubbleSort
O BubbleSort percorre o vetor N vezes, e a cada passagem ele joga no topo o maior elemento da sequência. Pelo fato de ter que percorrer o vetor diversas vezes, ele ficou em último lugar em todos os testes realizados e é recomendado apenas se a quantidade de dados for muito pequena, pois, apesar de ser considerado o menos eficiente nos testes realizados, é o menos complexo para se implementar.

SelectionSort
O algoritmo de ordenação por seleção percorre um vetor encontrando repetidamente o elemento mínimo da parte não classificada e colocando o item no início. O algoritmo mantém dois subarrays em um determinado array:

- Array que já foi ordenado.
- Array restante não foi ordenado.

Em cada iteração, o elemento mínimo do subarray não ordenado é selecionado e movido para o subarray ordenado.
O Selection Sort não é considerado um algoritmo eficiente para grandes entradas de dados. Há alternativas melhores para esses casos, como o Quicksort é o Merge Sort..

InsertionSort
É um algoritmo simples que funciona de maneira semelhante à maneira como classificamos as cartas de baralho. O vetor é dividido em dois: um ordenado e outro não ordenado. Os valores não ordenados selecionados são colocados na posição correta no vetor ordenado.
O InsertionSort é mais adequado quando se trata de uma pequena quantidade de dados, principalmente se existir a possibilidade desses dados já virem ordenados ou parcialmente ordenados.

QuickSort
É um algoritmo eficiente de ordenação também por divisão e conquista. Apesar de ser da mesma classe de complexidade do Mergesort e do Heapsort, o Quicksort é considerado o mais veloz deles já que suas partes divididas são menores. Também, por esse motivo, ele pode ser o mais complexo para implementar.

MergeSort
O MergeSort é o mais adequado para quantidades grandes de dados aleatorizados ou ordenados de forma decrescente. Dados ordenados ou parcialmente ordenados levam mais tempo nesse algoritmo levando em consideração o seu pilar (Dividir e Conquistar), que, irá desordenar os dados (pelo fato de dividir) e ter que ordenar novamente.

Todos os dados referente aos testes estão disponíveis em: github.com/chalestristian

4. Observe o algoritmo representado a seguir:

```
public void ordena(int v[]){
        for (int i = 1; i < v.length; i++) {
             aux = v[i];
             j = i;
            while ((j > 0) && (v[j - 1] > aux)) {
                 v[j] = v[j - 1];
                 j -= 1;
            }
            v[j] = aux;
        }
}
```

Esse algoritmo é o de: C

a. ShellSort
b. BubbleSort
c. InsertionSort
d. SelectionSort