UFMA - Universidade Federal do Maranhão
Componente Curricular: Estrutura de Dados II
Alunos: João Victor Abreu Machado e André Luiz Ribeiro de Araujo Lima

Introdução

Este projeto envolve a criação e implementação de algoritmos para responder a quatro questões apresentadas pelo professor em sala de aula, que compõem 50% da pontuação total da primeira avaliação. Foi utilizado o recurso de tipos genéricos da linguagem Java.

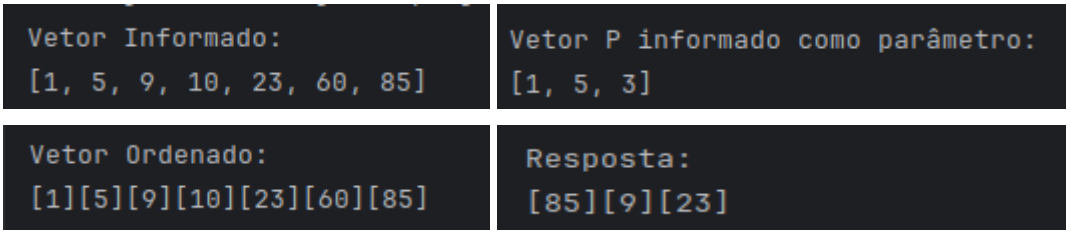
Como resultado deste projeto, foram desenvolvidas quatro classes distintas para abordar cada uma das questões propostas. Além disso, uma classe separada foi criada exclusivamente para os algoritmos de ordenação implementados. A classe principal é responsável por coordenar o funcionamento geral do programa.

Questão 1

Primeiramente foi necessário criar um algoritmo HeapSort Genérico, com isso as funções de heapify e buildHeap também foram implementadas utilizando a classe genérica. Após a ordenação do vetor, os K maiores elementos estão no final, com isso, conseguimos calcular e salvar os elementos desejados. Não foi possível obter o comportamento assintótico O(nlogk) pedido pelo professor e o melhor que conseguimos chegar foi O(nlogn).

```
no usages   joao victor
public  T[] KMaiores(T[] v1, int[] P) {

    heapSort(v1);
    int tamP = P.length;
    T[] vetRes = (T[]) new Number[tamP];
    for (int i = 0; i<tamP; i++) {
        vetRes[i] =  v1[v1.length - P[i]];
    }
return vetRes;
}
```

No exemplo desejamos saber o primeiro, quinto e terceiro maiores elementos de um vetor v1 informado.



Questão 2

O primeiro passo para resolver esse problema é concatenar e ordenar os dois vetores que devem ser passados como parâmetros. Para isso, implementamos um Merge Sort genérico, com comportamento assintótico O(nlogn). Uma vez que o vetor concatenado esteja ordenado, agora é possível calcular sua mediana, para isso, devemos nos atentar a 2 casos: quando o tamanho é ímpar e quando o tamanho é par. Nos casos de tamanho par quando os vetores informados são de números, a mediana é calculada pela média da soma dos elementos vetor[meio] + vetor[meio+1], se os vetores informados são de tipo String, a mediana é calculada pela concatenação dos mesmos. Já nos casos de tamanho ímpar, a mediana é simplesmente o elemento vetor[meio].
O comportamento assintótico é O(nlogn), uma vez que usamos o Merge Sort na ordenação do vetor.

```java
public class Problema2<T> {
    no usages  new *
    public T mediana(T[] v1, T[] v2) {
        int tam = v1.length + v2.length;

        //PRIMEIRO PASSO: Vamos criar um novo vetor que contenha o v1 e o v2
        T[] vetor = (T[]) new Object[tam];
        System.arraycopy(v1,  srcPos: 0, vetor,  destPos: 0, v1.length);
        System.arraycopy(v2,  srcPos: 0, vetor, v1.length, v2.length);

        //SEGUNDO PASSO: Agora é necessário ordenar o vetor para encontramos a mediana
        Sorting ordena = new Sorting();
        ordena.sort(vetor,  l: 0,  r: tam - 1);    //O(nLogn)
        System.out.println("vetor ordenado: ");
        for (T i : vetor) {
            System.out.print(i.toString());
            System.out.print(" ");
        }

        //TERCEIRO PASSO: Agora precisamos calcular a mediana, para isso:
        Object mediana;
        int meio = tam / 2 - 1;
        if (tam % 2 == 0) {
            if (vetor[meio] instanceof String) {
                mediana = (Object) ((vetor[meio].toString()).concat(vetor[meio + 1].toString()));
            } else {
                mediana = (Object) ((Integer.parseInt(vetor[meio].toString()) + (Integer.parseInt(vetor[meio + 1].toString()))) / 2);
            }

        } else {
            mediana = (Object) vetor[meio + 1];
        }

        return (T) mediana;

    }
}
```

Na execução com números temos, com tamanho ímpar:

```
Vetores Informados:
[2, 60, 35, 41, 25, 26, 69, 16, 3, 5]
[79, 78, 25, 35, 44, 23, 99, 55, 75, 68, 18]
```

```
vetor ordenado:
2 3 5 16 18 23 25 25 26 35 35 41 44 55 60 68 69 75 78 79 99
Mediana: 35
```

Já na execução com Strings, com tamanho par:

```
Vetores Informados:
[d, g, a, z, b]
[p, z, c, t, s]
```

```
vetor ordenado:
a b c d g p s t z z
Mediana: gp
```

Questão 3

Letra A

A resolução do problema consiste ordenar o vetor com insertion sort implementado com generics, depois disso, fazemos uma verificação percorrendo todo o vetor com objetivo de verificar se a diferença de um elemento na posição i e um elemento na posição j tem uma diferença igual ao parâmetro T, passado no cabeçalho da função, também com custo O(n²). Uma vez encontrada a diferença, salvamos os elementos de posição i e j, que retornaremos como resultado do algoritmo.
O comportamento assintótico desse algoritmo é O(n²) devido ao uso do insertionsort no momento de ordenação do vetor.

```java
public T[] distanciaTa (T [] vetor, int D) {
    //Solução O(n²)
    T[] vetorRes = (T[]) new Number[2];
    int tam = vetor.length;

    //instanciando um objeto da classe Sorting para utilizar os algoritmos de ordenação
    Sorting ordena = new Sorting();

    //ordena o vetor utilizando insertionSort
    ordena.insertionSort(vetor);

    //mostra o vetor ordenado
    System.out.println("vetor A ordenado:");
    for (T i: vetor) {
        System.out.print(i.toString());
        System.out.print(" ");
    }
    System.out.println();

    //percorre o vetor utilizando 2 for aninhados
    //subtrai cada elemento com todas as combinações possíveis e guarda o valor na variável chave
    for (int i = 0; i < tam; i++) {
        for (int j = i + 1; j < tam; j++) {
            Number aux = (Number) ((Integer.parseInt(vetor[i].toString())) - (Integer.parseInt(vetor[j].toString())));
            T chave = (T)aux;

            //se a chave for negativa, multiplica por -1 para corrigir e voltar a ser positiva
            if (Integer.parseInt(chave.toString()) < 0){
                Number aux2 = (Number) ((Integer.parseInt(chave.toString())) * -1);
                T novaChave = (T) aux2;

                //testa se a chave é igual a distância desejada
                //se sim, guarda os dois elementos correspondentes a essa distância no vetorRes
                if (((Integer.parseInt(novaChave.toString())) == D)) {
                    vetorRes[0] = vetor[i];
                    vetorRes[1] = vetor[j];
                }
            }
```

```
                else {
                    if (((Integer.parseInt(chave.toString())) == D)) {
                        vetorRes[0] = vetor[i];
                        vetorRes[1] = vetor[j];
                    }
                }
            }
        }

        return vetorRes;
    }
```

T = 3

```
Vetor Informado:
[1, 9, 0, 4, 6]
```

```
vetor A ordenado:
0 1 4 6 9
```

```
Resposta:
[6][9]
```

Letra B

A resolução do problema consiste em ordenar o vetor com Merge Sort implementado com generics, depois disso fazemos uma verificação percorrendo todo o vetor e aplicando uma busca binária (O(logn)) para ver se conseguimos encontrar a diferença entre um elemento[i] e o parâmetro T, passado no cabeçalho da função. Caso encontremos, isso quer dizer que achamos os dois elementos com distância T que estávamos procurando, salvamos e retornamos como resultado do algoritmo.

O comportamento assintótico desse algoritmo é O(nlogn) devido ao uso do mergesort no momento de ordenação do vetor.

```
public T[] distanciaTb (T[] vetor, int D) {

    //Solução O(nlog(n))
    T [] vetorRes = (T[])new Number[2];
    int tam = vetor.length;
    System.out.println("\nVetor B ordenado:");

    //instancia um objeto da classe Sorting para utilizar os algoritmos de ordenação
    Sorting ordena = new Sorting();

    //ordena utilizando mergeSort
    ordena.sort(vetor, l: 0, r: tam-1);
    for (T i: vetor)
        System.out.printf(i.toString()+" ");
    System.out.println();

    for (T j : vetor) {
        //para cada elemento do vetor, verifica qual seria o elemento corresponde para a distância pedida
        //e guarda na variável chave
        Object aux = (Object)(D - (Integer.parseInt (j.toString())));
        T chave = (T) aux;

        //se a chave for menor que 0, multiplica por -1 para corrigir e voltar a ser positivo
        if ((chave instanceof Integer) && (Integer)chave < 0) {
            Integer novaChave= Integer.parseInt(chave.toString());
            novaChave *= -1;
        }

        //utiliza busca binária para ver se a chave está no vetor
        if (ordena.buscaBinaria(vetor, chave)) {
            vetorRes[0] = j;
```

T = 3

```
Vetor Informado:
[3, 3, 8, 0, 9]
```

```
Vetor B ordenado:
0 3 3 8 9
```

```
Resposta:
[3][0]
```

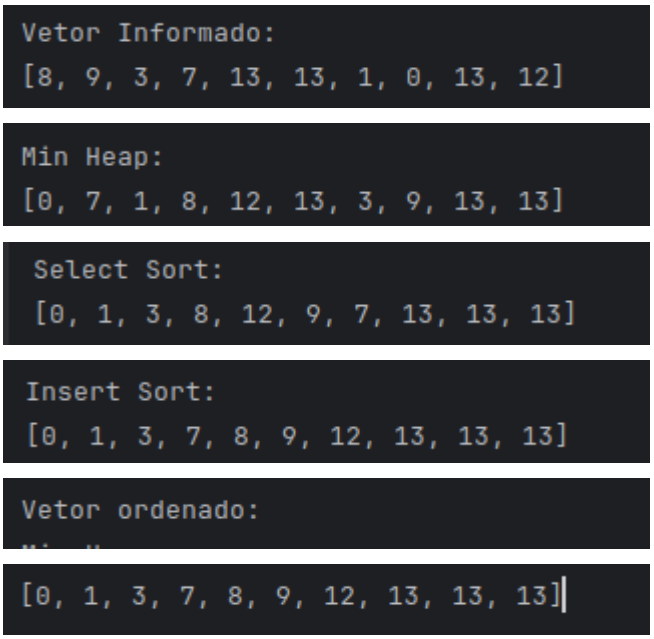
Questão 4

Para esse problema, nosso objetivo era criar um algoritmo denominado BHSI-Sort, combinando BuildHeap + Selection Sort + Insertion Sort. Para usar esse algoritmo, o usuário deve inicialmente construir o Heap (Max ou Min).Em seguida, ordenar as extremidades definidas em %E do tamanho do vetor usando o SelectSort. Por último, ordenar os elementos centrais usando o InsertSort.
Na solução, dividimos o array em 3 partes, para que possamos realizar as operações que foram solicitadas. Salvando os índices das posições que dividem o vetor, conseguimos determinar onde iremos aplicar o Select Sort e o Insert Sort.

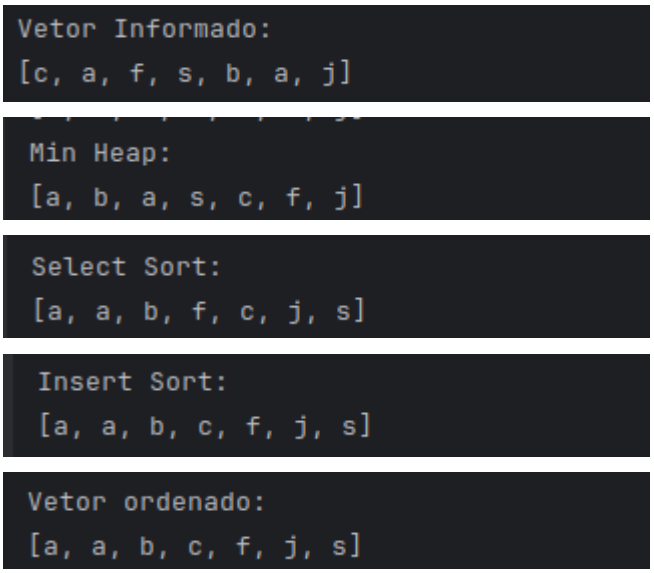
```
2 usages  joao victor
public T[] BHSISort(T[] v1){

    int endIndex = ((v1.length / 3) - 1 ); // Calcula o índice final para o primeiro Selection Sort
    int startInsert = endIndex + 1; // Índice inicial para Insertion Sort

    int startIndex = ((v1.length - endIndex) - 1); // Calcula o índice inicial para o segundo Selection Sort
    int endInsert = startIndex-1; // Índice final para Insertion Sort

    // Construir um heap máximo
    buildMinHeap(v1, v1.length);

    // Ordenar as extremidades com Selection Sort
    selectionSort(v1, endIndex, startIndex);

    // Ordenar o meio com Insertion Sort
    insertionSort(v1, startInsert, endInsert);



    return v1;
}
```

Exemplo de execução do algoritmo:

Com inteiros:



Com string:



Conclusão

Com a realização desse trabalho pode-se notar a importância do tipo generics na criação de códigos em java, sua flexibilidade e possibilidade de reutilização de código sem tantos problemas.