Análise e Projeto de Algoritmos
Trabalho 02

Maurício El Uri - 161150897

[1]Engenharia de Software – Universidade Federal do Pampa (UNIPAMPA)
Alegrete – RS – Brazil

mauriciom.eluri@gmail.com

1. Informações gerais sobre o trabalho

O algoritmo escolhido para desenvolvimento foi o algoritmo *1.6. Onde está a bolinha de gude?*. O software foi desenvolvido na linguagem Java.

Para compilar e executar a aplicação, basta importar o projeto em uma IDE de sua preferência, ou compilar o projeto manualmente, através do compilador do Java. Recomendamos que o projeto seja importado no Netbeans, pois foi desenvolvido nesta IDE e já temos os arquivos do projeto na IDE para facilitar a implantação. Para implantar o projeto no Netbeans, basta clicar em "$File > OpenProject...$" e selecionar o arquivo do projeto, como mostra na figura 1.

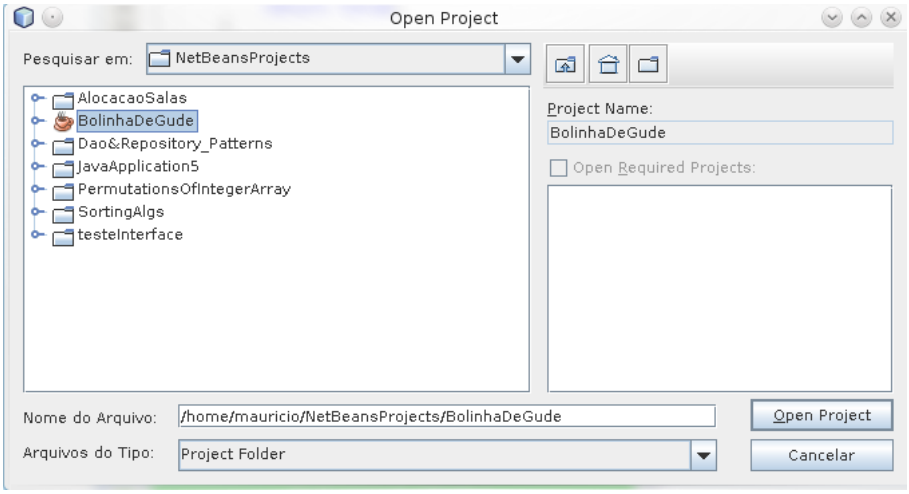

Figura 1. Importação do projeto no Netbeans

2. Descrição da aplicação

Nesta aplicação, o único arquivo executável é o arquivo TextualGame.java, o qual é a implementação do jogo como mostrado na especificação. Apesar de os outros arquivos não serem executáveis, todos podem ser testados através das classes de teste criadas. O software foi muito bem testado, principalmente por ter sido desenvolvido através do paradigma de desenvolvimento TDD.

2.1. Classe JogoBolinha

A classe **JogoBolinha** é uma classe abstrata que contém métodos que as duas implementações do jogo utilizarão. Ela possui 3 listas, a lista de bolinhas, consultas e

de resultados. A lista de resultados contém um array com o resultado para cada consulta, onde a posição [0] é a consulta e a posição [1] contém o índice de onde o resultado foi encontrado na lista de bolinhas. Caso a consulta não tenha sido encontrada, é salva no array de resultado como [-1].

Esta classe contém os métodos $setBolinha$ e $setConsulta$, que inserem novas bolinhas e consultas na lista, um a um. Estes métodos utilizam a classe $setValue$ para validar se os valores estão entre 0 e 10000. Caso os valores estejam fora do permitido, uma excessão é lançada.

Também temos a função protegida $setaResultado$, que será utilizada para criar um array com a consulta e os resultados e inserir o array na lista de resultados.

Por fim, para retornar os dados, temos as funções $getResultados$ que retorna a lista de resultados, a função $getBolinhas$ que retorna a lista de bolinhas e a função $getConsulta$ que retorna a lista de consultas. Também temos uma função chamada $getResultadosTxt$, que retorna os resultados em formato textual, que é usada na implementação do jogo em prompt de comando.

2.2. Classe JogoBF

A classe **JogoBF** é a extensão da classe **JogoBolinha**, que utiliza o paradigma de programação Força Bruta para encontrar os resultados. Ela tem apenas 2 funções, a função $executa$, que para cada item na lista de consultas, chama a função $executaConsulta$, passando como parâmetro a própria consulta. E temos a função $executaConsulta$, que recebe uma consulta como parâmetro e processa os resultados do jogo. Ela define o resultado inicial como -1 (não encontrado) e verifica toda a lista de bolinhas se o existe alguma bolinha com o mesmo valor que o recebido por parâmetro. Caso sim, a variável resultado é atualizada para o valor do índice da bolinha encontrada na lista e é chamada a função $setaResultado$ para inserir os valores no array de resultados.

2.3. Classe BubbleSort

A classe **BubbleSort** é a implementação de um algoritmo BubbleSort para ordenação da lista de bolinhas, utilizado na classe **JogoDC**. É importante ressaltar que o algoritmo de BubbleSort não foi escolhido por alguma razão específica, mas por este já ter sido desenvolvido anteriormente na disciplina. Esta classe contém uma lista de números inteiros, que é a lista que será ordenada. A lista deve ser passada já na construção da classe. Caso a classe receba uma lista nula, é retornado um exception.

A classe possui uma função $sort$, que é responsável por varrer a lista de trás pra frente, chamando o método $moveMaiorParaOFinal$ para cada item encontrado.

A função $moveMaiorParaOFinal$ recebe o valor atual que a formatação da lista está. Ele percorre a lista até a posição inicial e então substitui os valores da posição final recebida pelo maior valor encontrado, através da função $trocaValoresList$.

A função $trocaValoresList$ recebe dois índices na lista e é responsável por trocar seus valores.

Por fim, temos a função $getList$ que retorna a lista.

2.4. Classe JogoDC

A classe **JogoDC** é a uma extensão da classe **JogoBolinha**, e utiliza o paradigma de programação Divisão e Conquista, através de uma busca binária.

Esta classe, possui a função $executa$, que executa a busca utilizando o método divisão e conquista. Primeiramente, é criada uma cópia da lista de bolinhas, que é então, ordenada. Logo, para cada consulta, é realizada uma busca binária na lista de bolinhas ordenada. Esta função utiliza o método $setaResultado$ da inserir os resultados na lista de resultados.

Para ordenar as bolinhas, temos a função $ordenaBolinhas$, que utiliza a classe **BubbleSort** para a ordenação da lista. Para isto inserimos um clone da lista de bolinhas na classe de ordenação, mandamos ela realizar a ordenação da lista, e retornamos a lista ordenada.

Por fim, temos a função recursiva $buscaBinaria$, que recebe como parâmetro uma lista de bolinhas, um valor mínimo e máximo, e a consulta a ser realizada. Primeiramente, é testado se o valor desejado está no meio da lista. Caso não, verificamos se o valor mínimo é maior ou igual ao valor máximo, o que significa que a consulta não foi encontrada na lista. Caso isto aconteça é retornado o resultado -1. Caso ainda tenha o que se procurar na lista, testamos se o número a ser consultado é maior ou menor que o que está no meio da lista. Caso for maior, chamamos o método novamente, passando apenas a metade maior da lista. Caso for menor, chamamos o método passando apenas a metade menor da lista.

2.5. Classe executável TextualGame

A classe executável **TextualGame** utiliza a classe **JogoBF** para executar o jogo de maneira textual, da forma como foi descrito no trabalho. Ela tem uma função chamada $recebeInt$, que recebe um valor e utiliza a função $validaEntrada$ para verificar se o valor inserido é válido. A função $recebeInt$ fica em loop até que o usuário insira um valor válido, e caso o valor inserido seja inválido também é exibida uma mensagem informando o erro.

Por fim, temos a função $main$ que processa o jogo da mesma maneira como está na descrição do trabalho.

2.6. Classes de teste

As 3 classes de teste testam a classe **BubbleSort**, a classe que implementa o jogo com força bruta **JogoBF**, e o jogo com divisão e conquista, **JogoDC**.

3. Análise da complexidade - JogoBF.java

3.1. Função executaConsulta

```
1 private void executaConsulta(int consulta) {
2     int resultado = -1;
3     for (int i = 1; i < getBolinhas().size(); i++) {
4         if (consulta == getBolinhas().get(i - 1)) {
5             resultado = i - 1;
6         }
```

```
    }
    setaResultado(consulta, resultado);
}
```

Complexidade:

C2 $\rightarrow 1$

C3 $\rightarrow n + 1$

C4 $\rightarrow n$

C5 $\rightarrow n$

C8 $\rightarrow 1$

$T(n) = C2 + C3(n + 1) + C4n + C5n + C8$

$T(n) = C3(n + 1) + C4n + C5n$

$T(n) = (C3 + C4 + C5)n + 1$

$T(n)$ é $\mathcal{O}(n)$

3.2. Função executa

```
public void executa() {
    for (int consulta : getConsultas()) {
        executaConsulta(consulta);
    }
}
```

Complexidade:

C2 $\rightarrow n + 1$

C3 $\rightarrow n * m$

$T(n) = C2(n + 1) + C3(n * m)$

$T(n)$ é $\mathcal{O}(n * m)$

Obs: A linha C3 tem complexidade $n * m$, o que se dá pela quantidade n de consultas vezes a quantidade m de bolinhas.

3.3. Complexidade total algoritmo

A complexidade é Big Oh da quantidade de consultas multiplicada pela quantidade de bolinhas inseridas: $\mathcal{O}(consultas * bolinhas)$.

4. Análise da complexidade - BubbleSort.java

4.1. Função trocaValoresList

```
private void trocaValoresList(int index1, int index2) {
    int temporario = list.get(index1);
    list.set(index1, list.get(index2));
    list.set(index2, temporario);
}
```

Complexidade:

C2 $\rightarrow 1$

C3 $\rightarrow 1$

C4 $\rightarrow 1$

$T(n) = C2 + C3 + C4$

$T(n)$ é $\mathcal{O}(1)$

4.2. Função moveMaiorValorParaOFinal

```
private void moveMaiorValorParaOFinal(int tamFinal) {
    for (int index = 1; index < tamFinal; index++) {
        if (list.get(index) < list.get(index - 1)) {
            trocaValoresList(index, index - 1);
        }
    }
}
```

Complexidade:

C2 $\rightarrow n + 1$

C3 $\rightarrow n$

C4 $\rightarrow n * 1$

$T(n) = C2(n + 1) + C3n + C4n$

$T(n) = (C2 + C3 + C4)n + 1$

$T(n)$ é $\mathcal{O}(n)$

4.3. Função Sort

```
public void sort() {
    for (int count = list.size(); count > 1; count--) {
        moveMaiorValorParaOFinal(count);
    }
}
```

Complexidade:

C2 $\rightarrow n + 1$

C3 $\rightarrow n * moveMaiorParaOFinal$

$T(n) = C2(n + 1) + C3(n * n)$

$T(n)$ é $\mathcal{O}(n^2)$

Obs: A complexidade da linha C3 se dá pela complexidade $\mathcal{O}(n)$ da linha C3, vezes a complexidade n da função $moveMaiorParaOFinal$.

4.4. Complexidade total algoritmo

O total da complexidade deste algoritmo pode ser observado na função $sort$, onde temos uma chamada de n vezes a função $moveMaiorParaOFinal$, que tem complexidade n. Como resultado, temos um algoritmo com a complexidade: $\mathcal{O}(n^2)$.

5. Análise da complexidade - JogoDC.java

5.1. Função ordenaBolinhas

```
1  private ArrayList<Integer> ordenaBolinhas() {
2      BubbleSort bs = new BubbleSort((ArrayList<Integer>)
3          getBolinhas().clone());
4      bs.sort();
5      return bs.getList();
6  }
```

Complexidade:

C2 $\rightarrow 1$

C4 $\rightarrow 1 * n^2$

C5 $\rightarrow 1$

$T(n)$ é $\mathcal{O}(n^2)$

Obs: A complexidade da linha C4 se dá por 1 vezes a função $sort$, que tem complexidade $\mathcal{O}(n^2)$.

5.2. Função buscaBinaria

```
1  private int buscaBinaria(ArrayList<Integer> bolinhas,
2          int minimo, int maximo, int consulta) {
3      int meio = ((maximo + minimo) / 2);
4      if (bolinhas.get(meio) == consulta) {
5          return getBolinhas().indexOf(
6                  bolinhas.get(meio));
7      }
8      if (minimo >= maximo) {
9          return -1;
10     } else if (bolinhas.get(meio) < consulta) {
11         return buscaBinaria(bolinhas, meio + 1, maximo, consulta);
12     } else {
13         return buscaBinaria(bolinhas, minimo, meio - 1, consulta);
14     }
15 }
```

Complexidade:

C3 $\rightarrow 1$

C4 $\rightarrow 1$

C5 $\rightarrow 1$

C8 $\rightarrow 1$

C9 $\rightarrow 1$

C10 $\rightarrow 1$

C11 $\rightarrow buscaBinaria * n/2$

C12 $\rightarrow 1$

C13 $\rightarrow buscaBinaria * n/2$

$T(n) = C3 + C4 + C5 + C8 + C9 + C10 + C12 + (C11 + C13)logn$

$T(n) = (C11 + C13)logn$

$T(n)$ é $\mathcal{O}(logn)$

5.3. Função executa

```
public void executa() {
    ArrayList<Integer> listaOrdenada = ordenaBolinhas();
    for (int consulta : getConsultas()) {
        int resultado = buscaBinaria(listaOrdenada, 0,
                    listaOrdenada.size() - 1, consulta);
        setaResultado(consulta, resultado);
    }
}
```

Complexidade:

C2 $\rightarrow 1 * ordenaBolinhas$

C3 $\rightarrow n + 1$

C4 $\rightarrow n * buscaBinaria$

C6 $\rightarrow n * setaResultado$

$T(n) = C2(n^2) + C3(n + 1) + C4(nlogn) + C6(n)$

$T(n) = C2(n^2) + C4(nlogn) + (C3 + C4)n + 1$

$T(n) = n^2 + nlogn + n + 1$

$T(n)$ é $\mathcal{O}(n^2)$

Obs: A linha $C2$ tem complexidade $\mathcal{O}(n^2)$ por conta da função $ordenaBolinhas$. Da mesma forma que a linha $C4$ tem complexidade $n * logn$ por conta de efetuar n chamadas à função $buscaBinaria$, que tem complexidade $\mathcal{O}(logn)$. A função não descrita $setaResultado$ tem complexidade $\mathcal{O}(1)$.

5.4. Complexidade total algoritmo

Primeiro vamos definir a complexidade total da subfunção buscaBinaria, que se dá por uma entrada n, que é subdividida em $n/2$ partes até encontrar o resultado. Sua complexidade final é $nlogn$, que é simplificada para $\mathcal{O}(nlogn)$.

A função sort do algoritmo BubbleSort foi definida como $\mathcal{O}(n^2)$. Logo, a função ordenaBolinhas, tem complexidade $\mathcal{O}(n^2)$.

Logo, a função executa, tem como complexidade $n^2 + nlogn$, o que é simplificado para $O(n^2)$, por conta da expressividade que a ordenação tem sob o algoritmo de busca binária.

6. Conclusão Analise Complexidade

Podemos concluir que a ordenação significa a maior parte do custo do algoritmo divisão e conquista. Porém, podemos perceber que sem a ordenação, a busca binária tem um custo

menor de processamento que o algoritmo de força bruta, tendo em vista que o força bruta tem complexidade $\mathcal{O}(n^2)$ enquanto que o de busca binária (sem contar com a ordenação) tem complexidade $\mathcal{O}(nlogn)$. Logo, em casos onde a ordenação não se torna necessária, o algoritmo de busca binária torna-se a melhor opção. Também é preciso ressaltar que existem algoritmos mais eficientes para fazer a ordenação que também podem ser utilizados no lugar do Bubble Sort.

Referências