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.

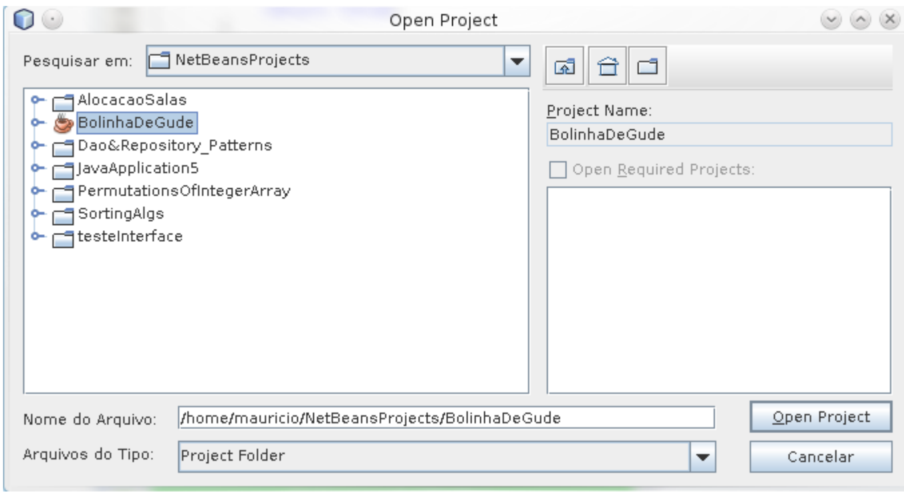

Figura 1. Importação do projeto no Netbeans

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. Também é importante ressaltar que o software está descrito em mais detalhes no próprio código, pois foi utilizado $Javadoc$ em todas as funções e métodos.

2.1. Classe JogoBolinha

A classe **JogoBolinha** é uma classe abstrata que contém métodos que as duas implementações do jogo utilizarão.

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, através de uma busca linear nas bolinhas.

2.3. Classe MergeSort

A classe **MergeSort** é a implementação de um algoritmo MergeSort para a ordenação da lista de bolinhas, utilizado na classe **JogoDC**.

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.

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.

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

```java
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         }
7     }
8     setaResultado(consulta, resultado);
9 }
```

Complexidade:

C2 $\rightarrow 1$

C3 $\rightarrow m + 1$

C4 $\rightarrow m$

C5 $\rightarrow m$

C8 $\rightarrow 1$

$$T(n) = C2 + C3(m+1) + C4m + C5m + C8$$

$$T(n) = C3(m+1) + C4m + C5m$$

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

$$T(n) \text{ é } \mathcal{O}(m)$$

3.2. Função executa

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

Complexidade:

C2 $\rightarrow n + 1$

C3 $\rightarrow n * m$

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

$$T(n) \text{ é } \mathcal{O}(n*m)$$

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 - MergeSort.java

4.1. Função merge

```java
private ArrayList<Integer> merge(ArrayList<Integer> listEsq,
            ArrayList<Integer> listDir) {

    ArrayList<Integer> listOrdenada = new ArrayList<>();
    ListIterator itrEsq = listEsq.listIterator();
    ListIterator itrDir = listDir.listIterator();

    while (itrEsq.hasNext() && itrDir.hasNext()) {
        int itemEsq = (int) itrEsq.next();
        int itemDir = (int) itrDir.next();
        if (itemEsq > itemDir) {
            listOrdenada.add(itemDir);
            itrEsq.previous();
        } else {
            listOrdenada.add(itemEsq);
            itrDir.previous();
        }
    }
    while (itrEsq.hasNext()) {
```

```
20            listOrdenada.add((int) itrEsq.next());
21        }
22        while (itrDir.hasNext()) {
23            listOrdenada.add((int) itrDir.next());
24        }
25        return listOrdenada;
26    }
```

Complexidade:

C4 $\rightarrow 1$

C5 $\rightarrow 1$

C6 $\rightarrow 1$

C8 $\rightarrow n+1$

C9 $\rightarrow n$

C10 $\rightarrow n$

C11 $\rightarrow n$

C12 $\rightarrow n$

C13 $\rightarrow n$

C14 $\rightarrow n$

C15 $\rightarrow n$

C16 $\rightarrow n$

C19 $\rightarrow 1+1$

C20 $\rightarrow 1$

C22 $\rightarrow 1+1$

C23 $\rightarrow 1$

C25 $\rightarrow 1$

$T(n) = C8 + C9 + C10 + C11 + C12 + C13 + C14 + C15 + C16(n)$

$T(n)$ é $\mathcal{O}(n)$

4.2. Função mergeSort

```
private ArrayList<Integer> mergeSort(int esq, int dir) {
    if (esq != dir) {
        int meio = ((esq + dir) / 2);
        ArrayList<Integer> listEsq;
        ArrayList<Integer> listDir;
        listEsq = mergeSort(esq, meio);
        listDir = mergeSort(meio + 1, dir);
        return merge(listEsq, listDir);
    }
    ArrayList<Integer> listOrdenada = new ArrayList<>();
    listOrdenada.add(list.get(esq));
```

```
    return listOrdenada;
}
```

Complexidade:

C2 $\rightarrow 1$

C3 $\rightarrow 1$

C4 $\rightarrow 1$

C5 $\rightarrow 1$

C6 $\rightarrow logn$

C7 $\rightarrow logn$

C8 $\rightarrow n(C6 + C7)$

C10 $\rightarrow 1$

C11 $\rightarrow 1$

C12 $\rightarrow 1$

$T(n) = C2 + C3 + C4 + C5 + C10 + C11 + C12 + C8n(C6 + C7)logn$

$T(n) = n * 2logn$

$T(n)$ é $\mathcal{O}(nlogn)$

4.3. Função Sort

```
public void sort() {
    if (!list.isEmpty()) {
        list = mergeSort(0, list.size() - 1);
    }
}
```

Complexidade:

C2 $\rightarrow 1$

C3 $\rightarrow nlogn$

$T(n) = C3nlogn + C2$

$T(n)$ é $\mathcal{O}(nlogn)$

4.4. Complexidade total algoritmo

O total da complexidade deste algoritmo pode ser observado na função $MergeSort$, onde temos 2 chamadas recursivas da função $mergeSort$ toda vez que a lista tiver mais do que 1 elemento. Além disso, temos a função $merge$ que une as 2 chamadas recursivas em uma lista ordenada. Como resultado temos um algorítmo com complexidade de tempo $\mathcal{O}(nlogn)$.

5. Análise da complexidade - JogoDC.java

5.1. Função ordenaBolinhas

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

Complexidade:

C2 $\rightarrow 1$

C4 $\rightarrow nlogn$

C5 $\rightarrow 1$

$T(n)$ é $\mathcal{O}(nlogn)$

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

5.2. Função buscaBinaria

```java
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

```
1 public void executa() {
2     ArrayList<Integer> listaOrdenada = ordenaBolinhas();
3     for (int consulta : getConsultas()) {
4         int resultado = buscaBinaria(listaOrdenada, 0,
5                     listaOrdenada.size() - 1, consulta);
6         setaResultado(consulta, resultado);
7     }
8 }
```

Complexidade:

C2 $\rightarrow 1 * ordenaBolinhas$

C3 $\rightarrow n + 1$

C4 $\rightarrow n * buscaBinaria$

C6 $\rightarrow n * setaResultado$

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

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

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

$T(n)$ é $\mathcal{O}(nlogn)$

Obs: A linha $C2$ tem complexidade $\mathcal{O}(nlogn)$ por conta da função $ordenaBolinhas$. Da mesma forma que a linha $C4$ tem complexidade $nlogn$ 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 MergeSort foi definida como $\mathcal{O}(nlogn)$. Logo, a função $ordenaBolinhas$, tem complexidade $\mathcal{O}(nlogn)$.

6. Conclusão Analise Complexidade

Podemos concluir que por conta de a ordenação ter a mesma complexidade de tempo que o algoritmo de busca binária, esta não teve impacto significativo em seu tempo de processamento. Logo, a busca binária teve complexidade de tempo $\mathcal{O}(nlogn)$. Já o algoritmo de força bruta tem complexidade $\mathcal{O}(n^2)$. Logo, em ambos os casos, mesmo com a ordenação, o algoritmo $JogoDC$ é mais eficiente.

Referências