RELATÓRIO T1 – MACAQUINHOS

Érick Berwanger Conde
22103041- 4

Escola Politécnica – PUCRS

13 de abril de 2023

RESUMO

Este relatório se trata de uma descrição do código, e desenvolvimento do mesmo, criado para resolver o problema apresentado no enunciado do Trabalho 1, da cadeira de Algoritmos e Estruturas de Dados II.

A questão a ser resolvida é a programação de um jogo de macacos, cocos e pedras, com suas poucas regras. A programação do dito jogo envolve vários dilemas de otimização, como decisões do que poderá ser útil e o que poderá ser descartado, visando um melhor funcionamento do código.

1. INTRODUÇÃO

O problema em questão trata-se de um jogo; com um número x de rodadas, um número y de macacos, sendo que cada macaco tem um número z de cocos, com mais uma variável de pedrinhas em cada coco. O jogo começa com o primeiro macaquinho distribuindo seus cocos para o macaco-alvo em questão, que é decidido por um número pré-determinado: existem duas possibilidades de macaco-alvo, uma para caso a quantidade de pedrinhas dentro do coco que será entregue for par, e outra para caso for ímpar, isso para cada coco que o macaco possuir. Tendo enviado todos os cocos para o macaco-alvo, o macaco seguinte repete o mesmo processo, até que todos os macacos tenham jogado, assim finalizando uma rodada e recomeçando o processo caso ainda falte ao menos uma rodada para ser jogada.

Junto ao enunciado, que explica as regras e apresenta um exemplo em miniatura do que teria que ser feito no trabalho em si, foram concedidos os casos em arquivos de texto, com cada caso sendo mais extenso que o anterior, contendo mais macacos e cocos, além de um número maior de rodadas. O programa teria então, que ler os arquivos e de alguma forma executar os comandos necessários para que o jogo acontecesse.

O que o enunciado deveria pedir, mas foi apresentado por fora, é que o código deve retornar apenas o macaco vencedor e sua quantidade de cocos no final de cada caso, e isso seria crucial na hora de arquitetar o código, já que com essa informação já seria possível determinar o que seria usado e o que não.

2. PRIMEIRA SOLUÇÃO

Estruturando pela primeira vez o código, sem saber o que seria necessário de fato para entregar como resultado do código, pois não constava no enunciado, foi criada uma classe “Macaco”, contendo o número do macaco, a quantidade de cocos que possuía, uma estrutura para guardar os tais cocos com o número de pedrinhas em cada, além do número do macaco que receberia os cocos, tanto em caso de cocos de quantidade par quanto de cocos de quantidade ímpar.

Essa classe também continha um ArrayList representando os cocos, com cada posição tendo suas pedrinhas salvas em números inteiros. A escolha do ArrayList para armazenar os cocos e pedrinhas foi decidida por lógica e facilidade na hora de manipular as variáveis; ao utilizar um ArrayList, era possível saber a quantidade de cocos que um macaco possuía apenas invocando o método “.size()” da estrutura, e seria possível mudar a quantidade de cocos e pedrinhas facilmente, utilizando dos métodos “.add()” e “.remove()”.

Mais tarde, a classe Macaco foi reestruturada utilizando a estrutura de pilhas da biblioteca Java; a classe Stack. Essa escolha foi feita pois a estrutura Stack possui um método de remoção que não usa nenhum índice como parâmetro, além de retornar o elemento removido na sua chamada, encurtando o código e também agilizando sua execução. Esse método se chama “.pop()”, e remove o último elemento da pilha, retornando o mesmo no final da execução, além de reformular o tamanho da pilha automaticamente.

Com o construtor finalizado, foi criado um método, cujo únicos parâmetros erma os números dos possíveis macacos-alvo, que era aplicado como o índice na lista de macacos criada na classe principal. Esse método removia todos os cocos do macaco, zerando seu contador de cocos e adicionando esse número ao contador do macaco-alvo. Antes de remover, conferia se a quantidade de pedrinhas era par ou ímpar e assim decidia qual macaco receberia o coco em questão.

Indo para a classe principal, foi reproduzido manualmente a situação de exemplo apresentada no enunciado, para que fosse possível realizar testes com o código. O número de rodadas eram 100.000 e haviam 6 macaquinhos, que foram criados manualmente e adicionados em um ArrayList. Seguem os parâmetros utilizados:

```
Fazer 100000 rodadas
Macaco 0 par -> 4 impar -> 3 : 11 :  178 84 1 111 159 22 54 132 201 51 44
Macaco 1 par -> 0 impar -> 5 : 9 :  80 82 10 83 98 31 56 84 53
Macaco 2 par -> 3 impar -> 4 : 7 :  65 194 35 132 191 202 62
Macaco 3 par -> 0 impar -> 4 : 3 :  121 10 162
Macaco 4 par -> 0 impar -> 5 : 5 :  16 110 125 113 35
Macaco 5 par -> 2 impar -> 0 : 8 :  120 25 20 134 166 100 157 159
```

A quantidade de cocos era o tamanho da pilha de cada macaco, que alterava conforme os cocos eram adicionados pelo método (de uma linha) “adicionarCoco(int pedrinhas)”, que simplesmente chamava o método “.add()” do Stack do Java, entrando o inteiro “pedrinhas” como parâmetro.

Com essa base criada, seria possível ver como o código estava funcionando. Apenas criando um laço simples utilizando a estrutura de “for” do Java, começando em 0 e com o limite de 100.000, foi possível notar uma demora na execução do código, o que não deveria ocorrer levando em conta que haviam apenas 6 macacos e uma pequena quantidade de cocos a serem manipulados.

Criando e testando com mais macacos (15) e mais cocos, fora notado um aumento significativo no tempo de execução do código, levando ao questionamento de que talvez não fosse isso que o enunciado pedia. Lendo o enunciado mais algumas vezes, chegou-se à conclusão de que era isso mesmo que deveria ser feito, porém mais uma dúvida surgiu: “O que exatamente o código deve retornar ao usuário?”. Perguntando aos colegas, foi informado que o código apenas precisaria retornar o macaco com mais cocos no final do jogo, e quantos cocos ele possuía, basicamente eliminando o que estava fazendo o código demorar para ser executado: a manipulação das pedrinhas.

Com essa descoberta, toda essa solução foi descartada e, semanas depois, retorno ao trabalho para tentar solucionar o código de maneira mais eficiente: Sem perder tempo, até mesmo tempo de execução, com informações que não seriam úteis para o output do código, e com a mente limpa, sem se prender de qualquer forma ao que tinha sido feito anteriormente.

3. SEGUNDA SOLUÇÃO

3.1 CLASSE CASO

Começando do zero e sabendo que, independentemente do que fora esperado de retorno do código, o input seria sempre pelos arquivos de texto fornecidos, o mais lógico a se fazer foi acessar os arquivos e montar uma estrutura para ser utilizada no código. Isso pouparia tempo na hora de testar o código, já que em vez de criar cenários em miniatura, estaria utilizando os casos providos junto com o enunciado do trabalho.

A ideia principal, que fora mantida até a última versão do código, foi a de transformar os arquivos em listas/arranjos: utilizando uma estrutura de matriz, seria possível acessar a linha, que condiz com número do macaco + 1, já que a primeira continha o número de rodadas a ser executado, utilizando o valor x da matriz, e suas informações pelo valor y. Ou seja, cada linha, exceto a primeira (x=0), que contém a instrução do número de rodadas, representaria um macaco. As informações do macaco (para quem seriam enviados os cocos, quantidade de cocos e quantidade de pedrinhas em cada coco) estariam divididas entre as posições da lista/arranjo contido na posição x da matriz.

Com isso em mente, era hora de decidir como esses dados seriam armazenados. A primeira ideia é a de usar um ArrayList guardando arranjos de String com os dados. Em pouco tempo foram notados dois problemas chaves;

1- Arranjos de String não seriam facilmente aumentados ou diminuídos, de acordo com a quantidade de cocos sendo recebidos e entregues a cada rodada;

2- Arranjos de String necessitariam de uma tradução para números inteiros a cada operação feita, podendo resultar na necessidade de mais estruturas para guardar os dados ou um enorme número de conversões durante a execução do código, comprometendo a eficiência do mesmo.

Com isso em vista, foi decidido que a melhor maneira de armazenar esses dados, afim de manipulá-los, seria com uma matriz composta por um ArrayList de ArrayLists de inteiros, já que todos os valores eram inteiros. Para fazer isso, foi usado um método simples e eficaz: primeiramente, todos os caracteres não-numéricos foram trocados por espaços. Logo após, utilizando o método Split, foram separados todos os números por posição, criando assim um arranjo de Strings para cada linha do arquivo, e adicionando ao ArrayList de linhas.

Tudo isso foi construído no construtor da classe “Caso”, que decidia por um switch case levando o número do caso como input, qual arquivo seria lido. E, por último, no método “getComandos()”, a matriz de ArrayLists é criada pela conversão dos dados já adquiridos em números inteiros.

*Adendo: apenas posteriormente foi lembrado que as pedrinhas não tinham grande importância, mas sim o fato de serem números pares ou ímpares, levando o código a ser modificado novamente, porém ainda usando ArrayLists em vez de arranjos básicos, tendo em vista que a classe principal já estava estruturada para a manipulação de ArrayLists, com todas as chamadas de métodos.

A principal modificação ocorreu no método “getComandos()”, que agora é chamado “getComandosAux()”, e serve de base para o novo getComandos(), que agora constrói uma nova matriz contendo as linhas e os macacos. A diferença dessa linha para a que era entregue anteriormente, é que nessa os macacos não teriam todas as informações dadas nos arquivos, mas somente as essenciais para a execução do jogo e descoberta do macaco vencedor. Essas informações são (numeradas por posição no ArrayList):

0 - Número do macaquinho

1 - Número do macaco-alvo, caso o coco tenha número par de pedrinhas

2 - Número do macaco-alvo, caso o coco tenha número ímpar de pedrinhas

3 - Quantidade total de cocos do macaco

4 - Quantidade de cocos com número par de pedrinhas do macaco

5 - Quantidade de cocos com número ímpar de pedrinhas do macaco

Os quatro primeiros dados são adquiridos copiando os dados já informados no ArrayList anterior. Os últimos dois dados, quantidade de cocos com números pares e ímpares de pedrinhas, por outro lado, são gerados no método. Isso é feito com um laço que percorre o ArrayList original, começando a partir da posição 4, onde a representação de cada coco, com suas respectivas representações de pedrinhas, começa, indo até o final da lista. Enquanto percorre essas posições o código confere se o número de pedrinhas é par, aumentando o contador caso seja. Contador esse, que é utilizado também para sabermos a quantidade de cocos com número de pedrinhas ímpares, apenas subtraindo o contador par, do número total de cocos.

3.2 CLASSE APP/MAIN

Já com os dados prontos para serem utilizados, a primeira coisa a ser feita é criar o caso, para se ter os dados manipuláveis na classe principal. Como a leitura personalizada do arquivo ocorre no próprio construtor, apenas cria-se um objeto “Caso” com o número do arquivo a ser lido, de 0 a 7. Depois criamos a matriz a ser manipulada, com o método getComandos do objeto Caso criado.

Para a quantidade de rodadas, teriam duas opções; ou ler a primeira linha de cada arquivo, que diz a quantidade de rodadas a serem feitas, ou perceber que a quantidade de rodadas é uma função onde f(x) = x * 100, sendo x a quantidade de macacos em cada caso. Com essa percepção, foi possível ter a quantidade de rodadas sem ler novamente o arquivo, ou reservar um ArrayList a mais com apenas uma posição contendo o valor.

O método “partida()” recebe como parâmetro a matriz com os macacos e suas informações essenciais e o número de rodadas. Tudo começa num laço que vai de 0 até o número de rodadas estipulado. Dentro desse laço, foi feito outro que percorre todas as linhas da matriz, enquanto executa a troca de cocos, caso haja ao menos um coco a ser enviado. Como os valores eram conhecidos e fixos para cada macaco, não houve necessidade de nenhum outro laço.

Por último, o método “macacoVencedor()” gera e retorna um arranjo de inteiros de tamanho 2, contendo o índice do macaco vencedor na posição 0, e a quantidade de cocos que o mesmo possui, na posição 1. Para conferir qual macaco termina com mais cocos, primeiro o método põe o primeiro macaco como o vencedor, e depois vai conferindo macaco por macaco até encontrar um macaco com mais cocos. Caso encontre, muda o número do macaco vencedor para o macaco que foi conferido e a quantidade de cocos pela quantidade de cocos do macaco em questão. Após, segue repetindo esse processo até que a lista de macacos/linhas seja percorrida por inteiro, retornando o macaco vencedor.

```
public static int[] macacoVencedor(ArrayList<ArrayList<Integer>> linhas){
    int macacoV = linhas.get(index:0).get(index:0);
    int qtddCocos = linhas.get(index:0).get(index:3);
    int[] vencedor = new int[2];
    for(int i = 1; i < linhas.size(); i++){
        if(linhas.get(i).get(index:3) > qtddCocos){
            macacoV = linhas.get(i).get(index:0);
            qtddCocos = linhas.get(i).get(index:3);
        }
    }
    vencedor[0] = macacoV;
    vencedor[1] = qtddCocos;
    return vencedor;
}
```

4. RESULTADOS

Os resultados a seguir foram conseguidos com a versão apresentada na segunda solução:

- **CASO 0**
  - 5.000 Rodadas
  - 50 Macacos
  - O macaco vencedor é o macaco nº9, com 2332 cocos

- **CASO 1**
  - 10.000 Rodadas
  - 100 Macacos
  - O macaco vencedor é o macaco nº20, com 15461 cocos

- **CASO 2**
  - 20.000 Rodadas
  - 200 Macacos
  - O macaco vencedor é o macaco nº38, com 74413 cocos

- **CASO 3**
  - 40.000 Rodadas
  - 400 Macacos
  - O macaco vencedor é o macaco nº36, com 145232 cocos

- **CASO 4**
  - 60.000 Rodadas
  - 600 Macacos
  - O macaco vencedor é o macaco nº177, com 230276 cocos

- **CASO 5**
  - 80.000 Rodadas
  - 800 Macacos
  - O macaco vencedor é o macaco nº20, com 182575 cocos

- **CASO 6**
  - 90.000 Rodadas
  - 900 Macacos
  - O macaco vencedor é o macaco nº589, com 433295 cocos

- **CASO 7**
  - 100.000 Rodadas
  - 1.000 Macacos
  - O macaco vencedor é o macaco nº144, com 581995 cocos

5. CONCLUSÃO

Esse trabalho apresentou um problema interessante, com diversas maneiras de ser resolvido. A medida que o código ia se estabelecendo, foi possível visualizar diferentes maneiras de chegar no objetivo de descobrir qual o macaco vencedor ao fim de cada caso.

A solução escolhida foi utilizar um sistema de matriz, composta por um ArrayList de ArrayLists, mais tarde foi percebido que poderiam ter sido usados arranjos simples, mas não tendo certeza do quão mais eficiente, ou menos, ficaria, optou-se por continuar com os ArrayLists. No final, o código executa todos os casos quase que instantaneamente, no computador utilizado (CPU: AMD Ryzen 7 5800X), enquanto a versão feita anteriormente chegava a demorar mais de dez minutos para executar caso número 4, demorando ainda mais para os seguintes.

O trabalho foi finalizado com um código genérico e otimizado o suficiente para que o mesmo problema que poderia ter sido resolvido em mais de 10 minutos, talvez mais de 10 horas até, executasse tudo conforme o planejado em menos questão de centésimos ou até mesmo milésimos de segundos. Isso leva a crer que o trabalho foi concluído com algumas dificuldades no caminho, mas certamente concluído com sucesso.

6. CÓDIGO