NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Game of Beans II

Análise e Desenho de Algoritmos

Clara Sousa Nº 58403 cjo.sousa@campus.fct.unl.pt
Henrique Ferreira Nº 55065 hjp.ferreira@campus.fct.unl.pt

April 14, 2021

1 Resolução do Problema

1.1 Apresentação da Função Recursiva

Apresentamos em seguida a definição da função recursiva S, desenvolvida com o objetivo de calcular a pontuação final (*Score*) de Jaba, no jogo Game Of Beans II, tendo em conta a sua abordagem optimal. Guardamos também, do *input*, o número máximo de pilhas de feijões que podem ser retiradas numa só jogada, a *depth D*, a sequência de pilhas *seq*, e a informação sobre qual o jogador que inicia o jogo. Note-se que a função S tem sempre em conta o número de elementos da subsequência atual de forma a garantir que, em qualquer chamada recursiva, o número máximo de pilhas de feijões possíveis de retirar é sempre superior ou igual a 1 e inferior ou igual a *min(D,j-i+1)*.

A função recebe então como argumentos a posição inicial i da sequência de pilhas de feijões, a posição final da mesma, j, e o jogador atual, p, sendo que este valor se encontra a 1 caso seja a vez da Jaba jogar, e 0 no caso de ser o Pieton.

$$S(i,j,p) = \begin{cases} 0 & j < i \\ 0 & j = i \quad e \quad p = 0 \\ seq[i] & j = i \quad e \quad p = 1 \\ S(i + PietonLeft(i, i + min(D,j\text{-}i\text{+}1) - 1, 0, maxLeftPieton), j, 1) & j > i \quad e \quad p = 0 \quad e \\ & maxLeftPieton \geq maxRightPieton \\ S(i, j - PietonRight(j - min(D,j\text{-}i\text{+}1) + 1, j, 0, maxRightPieton), 1) & j > i \quad e \quad p = 0 \quad e \\ & maxLeftPieton < maxRightPieton \\ max(maxLeftJaba, maxRightJaba) & j > i \quad e \quad p = 1 \end{cases}$$

Tendo:

$$maxLeftPieton = \max_{i \leq k \leq i + \min(D, j-i+1) - 1} (\sum_{n=i}^{k} (seq[n])) \tag{1}$$

$$maxRightPieton = \max_{j - \min(D, j-i+1) + 1 \leq k \leq j} (\sum_{n=k}^{j} (seq[n])) \tag{2}$$

$$maxLeftJaba = \max_{i \leq k \leq i + \min(D, j-i+1) - 1} (\sum_{n=i}^{k} (seq[n]) + S(k+1, j, 0)) \tag{3}$$

$$maxRightJaba = \max_{j - \min(D, j-i+1) + 1 \leq k \leq j} (\sum_{n=k}^{j} (seq[n]) + S(i, k-1, 0)) \tag{4}$$

Podemos ver que, para o caso da posição final da sequência ser inferior à inicial (ou seja, encontrando-nos numa sequência vazia), o valor de S será 0. No caso desta sequência apresentar apenas uma pilha de feijões e de ser Pieton a começar o jogo, Jaba fica também com 0 pontos; se for, contudo, Jaba a iniciar o jogo, então a sua pontuação final será aquela correspondente ao número de feijões na posição inicial, *seq[i]*.

Tendo explicado os casos base, passamos então para os casos em que ocorrem chamadas recursivas, nos quais a sequência dada no jogo apresenta já pelo menos duas pilhas de feijões.

Tratemos então do caso em que é a vez de Pieton jogar e, portanto, p encontra-se a 0. Sabemos que Pieton procura sempre a subsequência que maximiza a sua pontuação, tendo em conta apenas o estado atual do jogo; caso esta subsequência seja encontrada na parte esquerda da sequência, sendo o resultado em (1) superior ao de em (2), é chamada então novamente a função S, desta vez com uma nova posição inicial i (pois a nova sequência já não apresenta as pilhas à esquerda retiradas por Pieton), e passando a vez de jogar a Jaba, com $p = 1$. Caso a pontuação máxima se encontre à direita, sendo então o resultado em (1) inferior ao de em (2), a chamada recursiva é feita, então, passando uma nova posição final j (pois desta vez, as pilhas foram retiradas à direita e, portanto, ao final da sequência) e fazendo a mesma mudança de jogador. Caso Pieton encontre, tanto à esquerda, como à direita, o mesmo valor máximo de feijões, opta por escolher a subsequência da esquerda.

No caso de ser a Jaba a atual jogadora, é calculada então a melhor pontuação com base no máximo entre a pontuação máxima que Jaba pode obter retirando pilhas à esquerda, (3), e à direita, (4).

Para o cálculo da pontuação máxima possível de obter retirando pilhas na parte esquerda da sequência, (3), o raciocínio consiste em ir vendo, para cada subsequência possível de escolher, qual a pontuação máxima que será possível Pieton adquirir, após retirar tal subsequência (relembramos neste ponto que Jaba conhece a estratégia de Pieton e a usa no seu próprio algoritmo, neste preciso momento de cáculo) e, por sua vez, qual a pontuação que ela poderá adquirir com a nova sequência gerada após esta decisão futura de Pieton, e assim por adiante (recursivamente, até a sequência acabar). A chamada a S é feita, desta forma, variando o valor da posição inicial i da nova sequência gerada após a seleção de pilhas por parte de Jaba, mudando também o jogo para Pieton, $p = 0$, e somando a pontuação obtida fazendo esta escolha atual, o somatório com *seq[n]*, com a pontuação que Jaba obterá após Pieton jogar, *S(k + 1, j, 0)*.

O cálculo da pontuação máxima possível de obter retirando pilhas de feijões à direita, (4), é idêntico ao retratado relativamente à remoção à esquerda, com o pormenor de que, neste caso, a posição inicial i é fixa e, portanto, na chamada a S, é variada a posição final j, visto que é ao final da sequência original que as pilhas são retiradas.

Atente-se, neste ponto, que quando nos referimos à "remoção" de pilhas à esquerda e à direita, não nos referimos a uma literal remoção de partes da sequência, mas sim ao ajustamento das novas posições que designamos como inicial, i, e final, j.

Neste ponto torna-se relevante explicitar em que consistem as funções auxiliares *PietonLeft* e *PietonRight*.

$$PietonLeft(i, j, current, search) = \begin{cases} 1 & j \geq i \quad e \\ & current + seq[i] = search \\ 1 + PietonLeft(i+1, j, current + seq[i], search) & j > i \quad e \\ & current + seq[i] \neq search \end{cases}$$

$$PietonRight(i, j, current, search) = \begin{cases} 1 & j \geq i \quad e \\ & current + seq[i] = search \\ 1 + PietonRight(i, j-1, current + seq[j], search) & j > i \quad e \\ & current + seq[i] \neq search \end{cases}$$

Estas funções apresentam como argumentos as posições inicial e final de uma dada subsequência e assumem a existência de um valor *search*, que é também passado como argumento, que se trata da pontuação máxima possível de obter tendo em conta as subsqüências possíveis de fazer à esquerda, no caso de *PietonLeft*, e à direita, no caso de *PietonRight*.

A função recebe também o valor *current*, que se trata do máximo valor até então encontrado, e a ideia é ir percorrendo todas as subsequências possíveis e encontrar em que ponto, isto é, em que posição termina (no caso de *PietonLeft*), ou inicia (no caso de *PietonRight*), a subsequência cujo número total de feijões seja igual a *search*.

Caso cheguemos à última subsequência possível de realizar, e não tendo ainda encontrando o máximo *search*, assumimos que essa última subsequência é a que devolve esse valor máximo, pois se sabemos que esse valor existe, e não o tendo encontrado antes, só poderá estar nesta última subsequência.

1.2 Apresentação da Resolução em Programação Dinâmica

Com o fim de compreendermos como resolver o problema utilizando a programação dinâmica, decidimos começar por criar duas matrizes: uma para quando **Pieton** é o primeiro a jogar, **pieton**, e uma outra para quando é **Jaba** a primeira a jogar, **jaba**, tendo o índice de início da subsequência (indíce da esquerda), i, representando as linhas, e j, o índice de fim da subsequência (índice mais à direita), representando as colunas.

Decidimos também começar por averiguar o preenchimento destas matrizes com a sequência original:

2 -1 -1 -10 -1 20 -1 3

E com uma *depth* de 2, pois sabíamos, a partir do enunciado, o resultado do algoritmo de **Jaba** aplicado a estas condições.

Sabendo que, para ambas as matrizes, quando $j < i$, a pontuação de Jaba será sempre 0, preenchemos então estas entradas das matrizes com 0.

Conseguimos ver também que, relativamente às diagonais, isto é, para $j = i$, sendo **Pieton** o primeiro jogador, a pontuação de **Jaba** será de 0, pois **Pieton** levará a única pilha de feijões, enquanto que, seja o caso de ser **Jaba** a primeira jogadora, ela adicionará à sua pontuação o número de feijões nessa mesma pilha, *seq[i]*.

Tendo isso em mente, e encontrando-nos ainda nestes casos base, podemos preencher as matrizes **pieton** e **jaba** da forma em baixo apresentada, na qual os pontos de interrogação representam entradas por preencher.

i/j	1	2	3	4	5	6	7	8
1	0	?	?	?	?	?	?	?
2	0	0	?	?	?	?	?	?
3	0	0	0	?	?	?	?	?
4	0	0	0	0	?	?	?	?
5	0	0	0	0	0	?	?	?
6	0	0	0	0	0	0	?	?
7	0	0	0	0	0	0	0	?
8	0	0	0	0	0	0	0	0

Tabela 1: **pieton**

i/j	1	2	3	4	5	6	7	8
1	2	?	?	?	?	?	?	?
2	0	-1	?	?	?	?	?	?
3	0	0	-1	?	?	?	?	?
4	0	0	0	-10	?	?	?	?
5	0	0	0	0	-1	?	?	?
6	0	0	0	0	0	20	?	?
7	0	0	0	0	0	0	-1	?
8	0	0	0	0	0	0	0	3

Tabela 2: **jaba**

De que forma estamos, portanto, a percorrer a sequência inicial para preencher as matrizes? Já tendo preenchido os casos base, começamos na coluna em que *j=2* e vamos preenchendo as colunas seguintes, de "baixo para cima". Por exemplo, na coluna *j=4*, a sequência de preenchimento será: (3,4), (2,4), (1,4).

Vejamos, por exemplo, como preencher as posições (1,2) em cada uma das matrizes.

Pieton procura maximizar os seus pontos, pelo que, nesta jogada e jogando à esquerda, removeria apenas o 2, ficando **Jaba** com uma pontuação de -1. Jogando à direita, seriam removidos ambos o -1 e o 2, resultando para **Pieton** uma pontuação de 1. Sendo $2 > 1$, **Pieton** jogará então à esquerda retirando então a pilha com 2 feijões. Verificamos que pieton[1,2] = -1 e preenchemos, como tal, a entrada (1,2) com -1 na matriz **pieton**.

Nas mesmas circunstâncias, como seria o resultado de **Jaba**, sendo ela a jogar? Temos que jaba[1,2] = max(maxLeftJaba, maxRightJaba), com:

maxLeftJaba = max(seq[1] + pieton[2, 2], seq[1] + seq[2] + pieton[3, 2])

maxRightJaba = max(seq[1] + seq[2] + pieton[1, 0], seq[2] + pieton[1, 1])

Como sabemos já que pieton[2, 2] = 0, pieton[3, 2] = 0, pieton[1, 0] = 0 e pieton[1, 1] = 0, pois já preenchemos estas entradas na matriz **pieton**, temos que:

maxLeftJaba = max(2 + 0, 2 + (-1) + 0) = max (2,1) = 2

maxRightJaba = max(2 + (-1) + 0, -1 + 0) = max(1, -1) = 1

E, portanto, jaba[1,2] = max(2,1) = 2.

Podemos então ver que é possível calcular as posições seguintes de cada matriz com base nas posições anteriores das mesmas, sendo que, seja **Pieton** o primeiro jogador, podemos obter a pontuação final de **Jaba** acedendo à entrada na primeira linha e última coluna de **pieton**, e, seja **Jaba** a primeira, acedemos à mesma entrada em **jaba**.

2 Complexidade Temporal

No desenvolvimento da nossa solução, criámos uma classe *GameOfBeansII*, composta por dois métodos públicos principais e um construtor.

Quanto à complexidade temporal do **construtor**, vemos facilmente que esta é constante, $\Theta(1)$ - apenas atribuímos valores a variáveis e inicializamos a sequência e as matrizes com os tamanhos adequados.

No que toca ao método **addPile**, este apresenta também uma complexidade temporal constante, embora seja chamado P vezes, em que P corresponde ao número de pilhas de feijões da sequência em causa. Resulta, portanto, numa complexidade temporal total linear, $\Theta(P)$.

Por último, em **calcJabaFinalScore**, vários métodos auxiliares vão sendo empregues, pelo que devemos analisá-lo com mais cuidado, o que é feito em seguida. Podemos ver também que este método acaba por ter uma complexidade temporal de $O(P^2D)$:

- No primeiro ciclo, preenchemos apenas a diagonal da matriz **jaba**, ciclo esse executado P vezes, o que resulta numa complexidade de $\Theta(P)$.
- Podemos analisar o segundo ciclo observando a tabela apresentada na página seguinte e verificar que este apresenta uma complexidade de $O(P^2D)$.

Descrição	Operação	Complexidade
Percorrer Colunas das Matrizes este ciclo é executado P vezes	1	
Percorrer as Linhas de uma Dada Coluna nas Matrizes em cada iteração x da operação 1, este ciclo é executado x vezes; sabemos, portanto, que no total ele é executado 1 + 2 + ... + P vezes = $\frac{P(P-1)}{2}$	1.1	
Calcular Nova *depth*		$\Theta(1)$
Chamada a **getMaxPietonLeft** ciclo com operações de complexidades constantes, em que o número de execuções é no máximo D e no mínimo 1		O(D)
Chamada a **getMaxPietonRight** ciclo com operações de complexidades constantes, em que o número de execuções é no máximo D e no mínimo 1		O(D)
Preenchimento de **pieton**		$\Theta(1)$
Chamada a **maxLeftJaba** ciclo com operações de complexidades constantes, em que o número de execuções é no máximo D e no mínimo 1		O(D)
Chamada a **maxRightJaba** ciclo com operações de complexidades constantes, em que o número de execuções é no máximo D e no mínimo 1		O(D)
Preenchimento de **jaba**		$\Theta(1)$
Complexidade Total para Execução de 1.1		O(PD)
Complexidade Total para Execução de 1		$O(P^2D)$

Tabela 3: complexidade temporal do segundo ciclo de **calcJabaFinalScore**

3 Complexidade Espacial

Na nossa resolução guardamos 3 coisas: uma sequência para as pilhas de feijões e duas matrizes (**pieton** e **jaba**).

Estas estrutuuras são criadas no **construtor** e são dependentes do número de pilhas P passadas no *input*, de modo a que a sequência, sendo um *array*, apresenta uma complexidade espacial de $\Theta(P)$, e cada matriz apresenta uma complexidade de $\Theta(P^2)$.

O nosso programa apresenta, portanto, uma complexidade espacial de $\Theta(P^2)$.

4 Conclusões

4.1 Pontos Fortes

Quanto aos pontos fortes, referimos que o nosso código está bastante simplificado, legível e bem compartimentalizado.

4.2 Pontos Fracos

Relativamente aos pontos fracos, apontamos o facto de termos sentido a necessidade de recorrer a duas matrizes e não apenas a uma.

4.3 Alternativas Estudadas

Numa primeira abordagem à resolução do problema em programação dinâmica, começámos por assumir que seria mais intuitivo ter a sequência original a começar num índice i igual a 1, terminando no índice j, correspondente ao número de pilhas de feijões passados como *input*.

Ou seja, para uma sequência original:

2 -1 -1 -10 -1 20 -1 3

A sequência que utilizaríamos no nosso programa seria:

0 2 -1 -1 -10 -1 20 -1 3

Como consequência, cada matriz apresentaria uma linha e uma coluna a mais do que o apresentado atualmente na nossa solução. Tendo achado esta adição do zero desnecessária, e comprovando que não necessitávamos da mesma para uma correta solução, decidimos descartar tal alternativa.

4.4 Alternativas Por Estudar

No que toca à Programação Dinâmica, a opção de resolver o problema utilizando uma única matriz foi também colocada em cima da mesa. Porém, não conseguimos desenvolver um algoritmo com tal matriz, pelo que nos deixámos ficar pela resolução apresentada.

4.5 Possíveis Melhoramentos

Tendo já otimizado o máximo possível, acreditamos que poderíamos apenas ter arranjado uma solução com uma complexidade temporal mais favorável do que a atual.

5 Anexo

Deixamos anexo a este relatório o código do nosso programa, composto por 3 classes num mesmo pacote:

- *Main* - classe que faz a leitura dos testes *input*, apresentando o respetivo *output*;
- *BufferedScanner* - uma classe por nós desenvolvida logo no ínicio do semestre, que visa ler e processar *integers* com métodos do *BufferReader*;
- *GameOfBeansII* - classe que apresenta o algoritmo para a resolução do problema enunciado.

Bibliografia

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Third Edition.*

```java
import java.io.*;

/**
 * @authors Henrique Campos Ferreira - 55065 Clara Sousa - 58403
 */

public class Main {

    // Default value for buffer size
    private static final int BUFFER_SIZE = 10000;

    public static void main(String[] args) throws IllegalArgumentExceptio
n, IOException {

        // Creating the scanner
        BufferedScanner in = new BufferedScanner(new InputStreamReader(Sy
stem.in), BUFFER_SIZE);

        int numTests = in.nextInt();
        in.nextLine();

        // Going through all the tests
        for (int i = 0; i < numTests; i++) {

            int numPiles = in.nextInt();
            int depth = in.nextInt();
            in.nextLine();

            GameOfBeansII g = new GameOfBeansII(numPiles, depth);

            for (int j = 0; j < numPiles; j++)
                g.addPile(in.nextInt());

            in.nextLine();

            String player = in.nextLine();
            boolean firstPlayer = true;

            if (player.equals("Jaba"))
                firstPlayer = false;

            long result = g.calcJabaFinalScore(firstPlayer);

            System.out.println(result);
        }

        in.close();
    }
}
```

```java
import java.io.*;
import java.util.*;

/**
 * @authors Henrique Campos Ferreira - 55065 Clara Sousa - 58403 Mix in
 */

public class BufferedScanner implements Closeable {

    private static final String DELIMITER = " ";

    private BufferedReader reader;

    // Current line being read
    private String line;
    // Current position in the line being read
    private int currentLinePos;
    // Array of tokens
    private String[] tokens;
    // Current position in the tokens array
    private int currentTokensPos;

    // True if a new line is to be read
    private boolean nextLine;
    // True if a line has already been parsed
    private boolean tokenized;
    // True if there are no more lines to be read
    private boolean finished;
    // True if the scan is closed
    private boolean closed;

    /**
     * Creates a new BufferedScanner given the reader
     *
     * @param reader
     */
    public BufferedScanner(Reader reader) throws IOException {
        this.reader = new BufferedReader(reader);
        finished = false;
        closed = false;
        getLine();
    }

    /**
     * Creates a new BufferedScanner given the reader
     *
     * @param reader     - reader
     * @param bufferSize - size of the buffer
     * @throws IOException
```

```
     * @throws IllegalArgumentException
     */
    public BufferedScanner(Reader reader, int bufferSize) throws IOExcept
ion, IllegalArgumentException {
        this.reader = new BufferedReader(reader, bufferSize);
        finished = false;
        closed = false;
        getLine();
    }

    /**
     * Scans the next token of the input as an integer
     *
     * @return the integer scanned from the input
     * @throws IOException
     * @throws NoSuchElementException - if input is exhausted
     * @throws InputMismatchException - if the next token does not match
the Integer
     *                                regular
     * @throws IllegalStateException  - if this scanner is closed
     */
    public int nextInt() throws IOException, NoSuchElementException, Inpu
tMismatchException, IllegalStateException {
        if (closed)
            throw new IllegalStateException();

        if (finished || (nextLine && !getLine()))
            throw new NoSuchElementException();

        if (hasFinishedLine())
            throw new InputMismatchException();

        if (!tokenized)
            parseLine();

        try {
            int result = Integer.parseInt(tokens[currentTokensPos]);
            currentLinePos += tokens[currentTokensPos++].length() + 1; //
 add 1 to count the delimiter.
            return result;
        } catch (Exception ex) {
            throw new InputMismatchException();
        }
    }

    /**
     * Advances this scanner past the current line and returns the input
that was
```

```
     * skipped. This method returns the rest of the current line, excludi
ng any line
     * separator at the end. The position is set to the beginning of the
next line.
     * Since this method continues to search through the input looking fo
r a line
     * separator, it may buffer all of the input searching for the line t
o skip if
     * no line separators are present.
     *
     * @return the line that was skipped
     * @throws IOException
     * @throws NoSuchElementException - if input is exhausted
     * @throws IllegalStateException  - if this scanner is closed
     */
    public String nextLine() throws IOException, NoSuchElementException,
IllegalStateException {
        if (closed)
            throw new IllegalStateException();

        if (finished)
            return null;

        if (nextLine) {
            getLine();
            nextLine = true;
            return line;
        }

        nextLine = true;
        // Return the rest of the line
        if (!hasFinishedLine())
            return line.substring(currentLinePos);
        else
            return "";
    }

    /**
     * Scans the next token of the input as a String
     *
     * @return the String scanned from the input
     * @throws IOException
     * @throws NoSuchElementException - if input is exhausted
     * @throws IllegalStateException  - if this scanner is closed
     */
    public String next() throws IOException, NoSuchElementException, Ille
galStateException {

        if (closed)
```

```
            throw new IllegalStateException();

        if (finished || (nextLine && !getLine()))
            throw new NoSuchElementException();

        if (hasFinishedLine())
            throw new NoSuchElementException();

        if (!tokenized)
            parseLine();

        String result = tokens[currentTokensPos];
        currentLinePos += tokens[currentTokensPos++].length() + 1; // add
 1 to count the delimiter.

        return result;

    }

    /**
     * Checking if the current line has ended
     *
     * @return true if the current line has ended, false if otherwise
     */
    private boolean hasFinishedLine() {
        return currentLinePos >= line.length();
    }

    /**
     * Getting the tokens from a line (that is, strings separated by spac
es)
     */
    private void parseLine() {
        tokens = line.split(DELIMITER);
        currentTokensPos = 0;
        tokenized = true;
    }

    /**
     * Reading a new line
     *
     * @return true if there is new line to be read, false if otherwise
     * @throws IOException
     */
    private boolean getLine() throws IOException {
        line = reader.readLine();
        nextLine = false;
        tokens = null;
        tokenized = false;
```

```
        if (line == null)
            finished = true;
        else
            currentLinePos = 0;

        return !finished;
    }

    @Override
    public void close() throws IOException {
        reader.close();
        closed = true;
        line = null;
        tokens = null;
    }

}
```

```java
/**
 * @authors Henrique Campos Ferreira - 55065 Clara Sousa - 58403
 */

/*
 * This class presents the Game of Beans II, where the best score Jaba ca
n
 * achieve is found
 */

public class GameOfBeansII {

    /**
     * The sequence of piles.
     */
    private int[] seq;

    /**
     * The next pile to be added.
     */
    private int nextPile;

    /**
     * The maximum number of piles that can be removed in a single play.
     */
    private int depth;

    /**
     * The matrix to save the computed score of Jaba, when Pieton is the
first.
     * player.
     */
    private long[][] pieton;

    /**
     * The matrix to save the computed score of Jaba, when Jaba is the fi
rst player.
     */
    private long[][] jaba;

    /**
     * Creates a new Game of Beans for the specified number of piles and
depth.
     *
     * @param nPiles - The number of piles.
     * @param depth  - The maximum piles a player can take in a single pl
ay.
     */
```

```java
    public GameOfBeansII(int nPiles, int depth) {
        this.depth = depth;
        seq = new int[nPiles];
        nextPile = 0;
        pieton = new long[nPiles][nPiles];
        jaba = new long[nPiles][nPiles];
    }

    /**
     * Adds a pile of beans to the sequence.
     *
     * @param beans - The number of beans.
     */
    public void addPile(int beans) {
        seq[nextPile++] = beans;
    }

    /**
     * Calculates the final score of Jaba.
     *
     * @param firstPlayer - The first player: true for Pieton, false for
Jaba.
     * @return The final score of Jaba.
     */
    public long calcJabaFinalScore(boolean firstPlayer) {

        /*
         * Base Case: Filling up the diagonals of the matrix of when jaba
 is the first
         * player. Notice there is no need to fill in the entries where "
0" should be,
         * since Java does so automatically
         */
        for (int i = 0; i < seq.length; i++) {
            jaba[i][i] = seq[i];
        }

        for (int j = 1; j < seq.length; j++) {

            for (int i = j - 1; i >= 0; i--) {

                int updated_depth = Math.min(depth, j - i + 1);

                // Filling the matrix for Pieton
                long[] leftMaxPieton = this.getMaxLeftPieton(i, updated_d
epth);
                long[] rightMaxPieton = this.getMaxRightPieton(j, updated
_depth);
```

```
                if (leftMaxPieton[0] >= rightMaxPieton[0]) {

                    if (i + (int) leftMaxPieton[1] > j)
                        pieton[i][j] = 0;
                    else
                        pieton[i][j] = jaba[i + (int) leftMaxPieton[1]][j
];
                }

                else {
                    if (i > j - (int) rightMaxPieton[1])
                        pieton[i][j] = 0;
                    else
                        pieton[i][j] = jaba[i][j - (int) rightMaxPieton[1
]];
                }

                // Filling the matrix for Jaba
                jaba[i][j] = Math.max(this.maxLeftJaba(i, j, updated_dept
h), this.maxRightJaba(i, j, updated_depth));

            }

        }

        if (firstPlayer)
            return pieton[0][seq.length - 1];
        else
            return jaba[0][seq.length - 1];
    }

    /* Auxiliary Methods */

    /**
     * Returning the maximum number of beans Pieton can get by playing on
 the left,
     * along with the minimum number of piles needed to achieve such scor
e
     *
     * @param i - index to begin the search
     * @param d - depth
     * @return array of maximum score and minimum piles removed
     */
    private long[] getMaxLeftPieton(int i, int d) {
        long max = Long.MIN_VALUE;
        long sum = 0;
        long nPilesRemoved = 0;
        long[] result = new long[2];
```

```
        for (int k = i; k <= i + d - 1; k++) {
            sum += seq[k];
            if (sum > max) {
                max = sum;
                nPilesRemoved = k - i + 1;
            }

        }

        result[0] = max;
        result[1] = nPilesRemoved;

        return result;
    }

    /**
     * Returning the maximum number of beans Pieton can get by playing on
 the right,
     * along with the minimum number of piles needed to achieve such scor
e
     *
     * @param j - index to begin the search
     * @param d - depth
     * @return array of maximum score and minimum piles removed
     */
    private long[] getMaxRightPieton(int j, int d) {

        long max = Long.MIN_VALUE;
        long sum = 0;
        long nPilesRemoved = 0;
        long[] result = new long[2];

        for (int k = j; k >= j - d + 1; k--) {
            sum += seq[k];
            if (sum > max) {
                max = sum;
                nPilesRemoved = j - k + 1;
            }

        }

        result[0] = max;
        result[1] = nPilesRemoved;

        return result;
    }

    /**
```

```
     * Returning the maximum number of beans Jaba can get by playing on t
he left
     *
     * @param i - index where subsequence begins
     * @param j - index where subsequence ends
     * @param d - depth
     * @return maximum score
     */
    private long maxLeftJaba(int i, int j, int d) {

        long sum = 0;
        long max = Long.MIN_VALUE;

        for (int k = i; k <= i + d - 1; k++) {

            sum += seq[k];

            long current = sum;

            // Accessing possible positions
            if (k + 1 <= j)
                current += pieton[k + 1][j];

            if (current > max)
                max = current;
        }

        return max;
    }

    /**
     * Returning the maximum number of beans Jaba can get by playing on t
he right
     *
     * @param i - index where subsequence begins
     * @param j - index where subsequence ends
     * @param d - depth
     * @return maximum score
     */
    private long maxRightJaba(int i, int j, int d) {

        long sum = 0;

        long max = Long.MIN_VALUE;

        for (int k = j; k >= j - d + 1; k--) {

            sum += seq[k];
            long current = sum;
```

```
            // Accessing possible positions
            if (i <= k - 1)
                current += pieton[i][k - 1];

            if (current > max)
                max = current;

        }

        return max;
    }
}
```