ADA
Project 1 Report
Game of Beans

Jacinta Sousa 55075
João Bordalo 55697

April 2021

Problem Resolution

Number of piles: $1 \leq P \leq 200$
Game depth: $1 \leq D \leq min(P, 10)$
Size of the i^{th} pile: $(-100 \leq s_i \leq -1) \vee (1 \leq s_i \leq 100)$, for every i = 0, 1, ..., P-1

$score(i, j) = \sum_{k=i}^{j}(s_k)$
This function returns the score obtainable for the sequence s_i ... s_j.

For example, in the given sequence 2 6 -1 3, it would return 2+6+(-1)+3=10.

$pieton(i, j)$: function which calculates Pieton's choice in the given i, ..., j sequence of piles and returns the remaining sequence after his choice in the form (i,j).

For example, in the given sequence 2 6 -1 3, one considers 2 to correspond to index 0 and the last pile of beans, with 3 beans, corresponds to index 3. With depth 1, the result of pieton(0,3) would be (0,2) representing the subsequence 2 6 -1, for he chooses the rightmost pile.

$$\mathcal{J}(i,j,p) = \begin{cases} 0 & (i = j \wedge p = Pieton) \vee no\ piles \\ s_i & i = j \wedge p = Jaba \\ max(max_{0 \leq k < D}(score(i, i+k) + \mathcal{J}(i+k+1, j, Pieton), & \\ \quad max_{0 \leq k < D}(score(j-k, j) + \mathcal{J}(i, j-k-1, Pieton) & j > i \wedge p = Jaba \\ \mathcal{J}(pieton(i,j), Jaba) & j > i \wedge p = Pieton \end{cases}$$

$\mathcal{J}(i, j, p)$ denotes Jaba's score with the sequence s_i, s_{i+1}, ..., s_{j-1}, s_j with $(0 \leq i \leq P-1 \wedge i \leq j \leq P-1)$ and p $\in \{Jaba, Pieton\}$, representing the current player.

Temporal Complexity

Populating scores' matrix:	$O(P^2)$
Populating Pieton's matrix:	$O(2 * D * P^3) \equiv O(P^3)^*$
First loop (Base Case):	$\Theta(P)$
Second loop (General Case)*:	$O(2 * D * P^2) \equiv O(P^3)^*$
	$O(P^2 + P^2 + P + P^3) \equiv O(P^3)^*$

* For D≈P, $D * P^2$ results in a worst case complexity of $O(P^3)$. For each iteration, we must, in the worst case, go through every pile in the sequence.

For D much smaller than P (D $\ll$ P), however, D can be ignored, resulting in $O(P^2)$ complexity across the program.

Spacial Complexity

∗ **Score**
At first one creates a PxP matrix to save the scores, for each entry (i,j), representing the subsequence s_i ... s_j, one will save score(i,j).[a]
$= \Theta(P^2)$

∗ **Pieton**
A PxPx2 matrix to save Pieton's choices, for each entry (i,j), corresponding to the subsequence s_i ... s_j, one will save [k,w] representing the subsequence s_k ... s_w Pieton chose.[b]
$= \Theta(2P^2) \equiv \Theta(P^2)$

∗ **Jaba**
At last, one creates a PxP matrix where each entry (i,j) keeps Jaba's score for the subsequence s_i ... s_j.[b]
$= \Theta(P^2)$

$= \Theta(P^2 + P^2 + P^2) \equiv \Theta(P^2)$

[a]. As seen in the first line of **populateScores()**, in Code Annex (page 6).
[b]. Initialized in the first lines of **computeScore()**, in Code Annex (page 6).

Conclusion

∗ **Strong Aspects**:
Repeated calculations were avoided by storing values in matrices.

∗ **Weak Aspects**:
Two of the main matrices aren't totally filled, nearly half of the slots are empty (filled with 0).

∗ **Alternatives Studied**:
Attempted mapping the diagonals onto rows, leading to complicated logic and confusing code.
Initial solution didn't store previously calculated scores, leading to unnecessary calculations, i.e. for a P sized pile, P*P calls are required to fill the scores' matrix and the program did approximately $10P^2$ calculations.

∗ **Improvements**:
Better use of the memory allocated for the matrices.
Improve locality of reference (avoid filling the tables diagonally).

Code Annex

Main.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));

        int testCases = Integer.parseInt(input.readLine());

        for (int i = 0; i < testCases; i++) {
            solveTestCase(input);
        }

        input.close();
    }

    public static void solveTestCase(BufferedReader input) throws IOException {
        String[] tokens = input.readLine().split(" ");
        int nPiles = Integer.parseInt(tokens[0]);
        int gameDepth = Integer.parseInt(tokens[1]);

        String[] pileInput = input.readLine().split(" ");
        int[] pile = new int[nPiles];

        for (int i = 0; i < pileInput.length; i++) {
            pile[i] = Integer.parseInt(pileInput[i]);
        }

        String firstPlayer = input.readLine();

        GameOfBeans game = new GameOfBeans(pile, gameDepth, firstPlayer);

        System.out.println(game.computeScore());
    }
}
```

GameOfBeans.java

```
public class GameOfBeans {

    private int gameDepth;
    private boolean isJabaFirst;
    private int[][] scores;
    private int BEGIN;
    private int END;

    public GameOfBeans(int[] pile, int gameDepth, String firstPlayer) {
        this.gameDepth = gameDepth;
        this.isJabaFirst = firstPlayer.equals("Jaba");
        this.BEGIN = 0;
        this.END = pile.length;
        populateScores(pile);
    }

    private int[] Pieton(int i, int j) {
        int max = Integer.MIN_VALUE;
        int[] indices = new int[2];

        // Left-side
        for (int d = 0; d < this.gameDepth && i + d <= j; d++) {
            int choice = this.scores[i][i + d];
            if (choice > max) {
                indices[0] = i;
                indices[1] = i + d;
                max = choice;
            }
        }

        // Right-side
        for (int d = 0; d < this.gameDepth && i + d <= j; d++) {
            int choice = this.scores[j - d][j];
            if (choice > max) {
                indices[0] = j - d;
                indices[1] = j;
                max = choice;
            }
        }

        return indices;
    }

    private void populatePieton(int[][][] pieton) {
        for (int i = 0; i < this.END; i++) {
            for (int j = i; j < this.END; j++) {
                pieton[i][j] = this.Pieton(i, j);
            }
        }
    }
```

```
private int[] computeRemainingPile(int left_bound, int right_bound, int[][][]
    pieton) {

    int[] indices = pieton[left_bound][right_bound];

    // Pieton chose from the left
    if (left_bound == indices[0]) {
        if (indices[1] + 1 > right_bound) return null;
        return new int[]{indices[1] + 1, right_bound};
    }
    // Pieton chose from the right
    else {
        if (left_bound > indices[0] - 1) return null;
        return new int[]{left_bound, indices[0] - 1};
    }
}

private void populateScores(int[] pile) {
    this.scores = new int[pile.length][pile.length];

    for (int i = this.BEGIN; i < this.END; i++) {
        int score = 0;
        for (int j = i; j < this.END; j++) {
            score += pile[j];
            this.scores[i][j] = score;
        }
    }
}

public int computeScore() {

    int[][][] pieton = new int[this.END][this.END][2];
    int[][] jaba = new int[this.END][this.END];

    // Populate Pieton's matrix
    this.populatePieton(pieton);

    // If Pieton moves first we need to find his move
    if (!this.isJabaFirst) {
        int[] pietonChoice = pieton[0][this.END - 1];

        if (pietonChoice[0] == 0) {
            this.BEGIN = pietonChoice[1] + 1;
        } else {
            this.END = pietonChoice[0];
        }

        if (this.BEGIN >= this.END) {
            return 0;
        }
    }
```

```
        // Fill first diagonal with the pile values
        for (int i = this.BEGIN; i < this.END; i++) {
            jaba[i][i] = this.scores[i][i];
        }

        int max;
        int score;

        // Run through the matrix diagonally
        for (int difference = 1; difference < this.END; difference++) {
            for (int i = this.BEGIN; i < this.END - difference; i++) {
                int j = i + difference;

                max = Integer.MIN_VALUE;

                // Left-side
                for (int d = 0; d < this.gameDepth && i + d <= j; d++) {
                    score = this.scores[i][i + d];

                    // remainingPile is null if it's empty
                    int[] remainingPile = null;
                    if (i + d != j) remainingPile = this.computeRemainingPile(i +
                        d + 1, j, pieton);
                    score += (remainingPile == null) ? 0 :
                        jaba[remainingPile[0]][remainingPile[1]];

                    max = Math.max(max, score);
                }

                // Right-side
                for (int d = 0; d < this.gameDepth && i + d <= j; d++) {
                    score = this.scores[j - d][j];

                    // remainingPile is null if it's empty
                    int[] remainingPile = null;
                    if (i != j - d) remainingPile = this.computeRemainingPile(i, j
                        - d - 1, pieton);

                    score += (remainingPile == null) ? 0 :
                        jaba[remainingPile[0]][remainingPile[1]];

                    max = Math.max(max, score);
                }

                jaba[i][j] = max;
            }
        }

        return jaba[this.BEGIN][this.END - 1];
    }
}
```