GameOfBeansII

André Santos, nº55415
Tiago Teles, nº54953

Resolução do Problema:

score(L)(i) devolve a pontuação máxima que a Jaba consegue ganhar para uma subsequência de pilhas contínua de comprimento L a começar no índice i, quando é a primeira a jogar. Caso seja o Pieton o primeiro a jogar o problema é reduzido a um tratável pela função score.

$$score(L)(i) = \begin{cases} 0, & L = 0 \\ piles[i], & L = 1 \\ max(prefixScore(L, i, 1..k), suffixScore(L, i, 1..k)) & L \geq 2 \end{cases}$$

Onde:

$$prefixScore(L, i, k) = \sum_{t=1}^{k}(piles[i + t]) + score(L - k - pilesPieton(i + k, i + L))(i + k + prefixPilesPieton(i + k, i + L))$$

$$suffixScore(L, i, k) = \sum_{t=1}^{k}(piles[i + L - t]) + score(L - k - pilesPieton(i, i + L - k))(i + prefixPilesPieton(i, i + L - k))$$

Em que:
$k \leq min(L, D)$
k é o número de pilhas removidas pela Jaba.
pilesPieton(i, j) devolve o número de pilhas removidas pelo Pieton para uma subsequência de pilhas contínua a começar no índice i e a terminar no índice j.
prefixPilesPieton(i, j) é análoga mas contabiliza somente as pilhas removidas do prefixo da subsequência.

Em mais detalhe: o caso base em que L=0 justifica-se pela razão de que para um qualquer subjogo de comprimento 0 é impossível remover pilhas, e sendo a pontuação base 0, esta assim se manterá. Para o segundo caso base (L=1), a Jaba é forçada a jogar pois só há uma pilha a retirar, ficando então com a pontuação dela proveniente. No caso geral, a Jaba considera todas as jogadas que lhe são possíveis, escolhendo de seguida aquela cuja soma da pontuação que acabou de obter com a pontuação que obterá no subjogo que resta após a jogada do Pieton seja máxima.

Complexidade Temporal

Caso Pieton seja o primeiro:
- Jogada Pieton: Θ(D)
- Reatribuir piles: O(P)

Criar caches Pieton (criar duas matrizes com P*D de dimensão (ou (P-X)*D, caso o Pieton jogue primeiro, sendo X o número de pilhas por ele removidas)): Θ(1)

Preencher cache Pieton (preencher todas (com exceção das D últimas linhas, que são preenchidas parcialmente) as posições de memória das tabelas acima mencionadas, em que cada preenchimento tem custo constante): O(PD)

Criar tabela Jaba (criar uma matriz análoga às da cache do Pieton mas invertida): Θ(1)

Preencher tabela Jaba valores iniciais (preencher todas as posições de memória da tabela acima mencionada, em que cada preenchimento tem custo constante): O(PD)

Preencher tabela Jaba: $O(P^2D)$ (o erro cometido no cálculo ao assumir P>>D é por excesso. É aceitável pois segundo as restrições do jogo tem-se P≥D, P≤100 e D≤10 pelo que esta é uma aproximação representativa da maioria dos casos. Considerando o caso limite em que P=D e substituindo min(L,D) por L chegar-se-ia a $O(P^3)$, que é consistente com $O(P^2D)$, dado P=D)

$$\sum_{L=2}^{P}\sum_{i=0}^{P-L}\sum_{r=1}^{min(L,D)} 1 = \sum_{L=2}^{P}\sum_{i=0}^{P-L} min(L,D) = \sum_{L=2}^{P}(P-L+1)*min(L,D) \overset{P>>D}{\rightarrow} \sum_{L=2}^{P}(P-L+1)*D =$$

$$D*(\sum_{L=2}^{P}(P) - \sum_{L=2}^{P}(L) + \sum_{L=2}^{P}(1)) = D*(2P*\sum_{L=2}^{P}(1) - \sum_{L=2}^{P}(L)) = D*(2P*(P-1) - \frac{P(P-1)}{2}) =$$

$$D*(2P^2 - 2P - \frac{P^2-P}{2}) = D*(\frac{P^2-P}{2}) = O(P^2D)$$

Complexidade total: $O(P^2D)$

Nota: A complexidade é $O(P^2D)$ e não $\Theta(P^2D)$ pois nos casos em que o Pieton joga primeiro as pilhas que o mesmo remove na primeira jogada não são consideradas na iteração das estruturas de dados. Para os restantes casos a situação é análoga.

Complexidade Espacial

Jogo corrente: Θ(1)
Cache par (pilhas, pontuação) para jogadas a retirar pilhas do prefixo do Pieton: O(PD)
Cache par (pilhas, pontuação) para jogadas a retirar pilhas do sufixo do Pieton: O(PD)
Tabela função score(L)(i): O(PD)

Complexidade total: O(PD)

Nota: As complexidades são O(PD) e não Θ(PD) somente porque nos casos em que o Pieton joga primeiro as pilhas que o mesmo remove na primeira jogada não são consideradas na criação das estruturas de dados.

Conclusões

Em termos de complexidade temporal cremos que a nossa solução não possa ser melhorada pois a componente máxima é a das jogadas da Jaba e advém da definição do problema que a mesma tem de ter conhecimento acerca de todos os subjogos provenientes daquele a ser avaliado (de onde surge a componente P^2) e que para cada um destes jogos tem considerar a melhor de 2*D jogadas possíveis (de onde surge a componente D) pelo que se for necessário consultar cada uma destas alternativas a complexidade temporal terá de ser no mínimo da ordem de grandeza P^2D. É necessário consultar todas as alternativas pois as distribuições de pilhas a serem avaliadas pela função score são arbitrárias e as jogadas do Pieton delas dependentes, pelo que não há nenhum padrão geral que permita inferir a partir de um subjogo informação acerca dos restantes.

Em termos de complexidade espacial é de notar que é o uso de uma "matriz circular" que nos permite atingir O(PD). Esta possibilidade deve-se ao facto de no caso geral da função score ser apenas necessário ter acesso aos valores até 2*D linhas antes.
Tentou-se ainda recorrer a estruturas que armazenassem somente as jogadas do Pieton para um dado subjogo. Estas tentativas, porém, revelaram-se infrutíferas dado que a implementação ficava consideravelmente mais complicada e as complexidades esperadas eram quadráticas.
A solução final por nós adotada será provavelmente uma das melhores, dado que o preenchimento de cada posição das caches tem custo constante e o conjunto (prefixo, profundidade (D), pontuação, jogada) armazenado em cada entrada contém informação generalizável para qualquer variante do jogo em que o prefixo seja igual e o comprimento (P) seja maior ou igual à profundidade (D) (esta explicação é só válida para a cache de jogadas a retirar do prefixo mas no que toca à do sufixo a situação é análoga). Há entradas com valores duplicados, mas não vemos formas eficientes de as evitar.

Já em termos absolutos espaciais seria possível armazenar a tabela da Jaba numa "matriz circular" em que cada linha tem dimensão inferior à anterior em uma unidade, o que se traduziria na poupança de cerca de $D^2/2$ posições de memória. Apesar de ser fácil de implementar, por não ser uma poupança muito significativa e em prole da simplicidade, o anteriormente referido não acabou por se concretizar. Nas caches do Pieton seria possível aplicar algo semelhante para as últimas D linhas das matrizes.

Algo mais interessante na vertente temporal, ainda em termos absolutos, seria guardar a tabela da Jaba com a dimensão resultante do ajuste acima proposto num vetor unidimensional circular, recorrendo a uma função para converter os índices, o que melhoraria do uso da cache do processador, especialmente para valores de P altos durante as últimas iterações de preenchimento.

```java
import java.io.IOException;

public class Main {

    public static void main(String[] args) throws IOException {
        TurboScanner in = new TurboScanner(System.in);
        int testCases = in.nextInt();
        for (int i = 0; i < testCases; i++) {
            int pileN = in.nextInt();
            byte[] piles = new byte[pileN];
            int depth = in.nextInt();
            for (int p = 0; p < pileN; p++) {
                piles[p] = (byte) in.nextInt();
            }
            GameOfBeans.Player p = in.next().equals("Jaba") ? GameOfBeans.Player.JABA :
GameOfBeans.Player.PIETON;
            GameOfBeans gb = new GameOfBeans(p, depth, piles);
            System.out.println(gb.score());
        }
    }
}

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
class TurboScanner{
   BufferedReader br;
   public TurboScanner(InputStream in){
      br = new BufferedReader(new InputStreamReader(in));
   }

   public int nextInt() throws IOException {
      return Integer.parseInt(next());
   }
   public String next() throws IOException{
      int i = 0;
      char ascii = '_';
      char[] chars = new char[13];
      while (ascii != ' ' && ascii != '\n') {
         ascii = (char) br.read();
         chars[i++] = ascii;
      }
      return new String(chars).trim();
   }
   public String nextLine() throws IOException{
      return br.readLine();
   }
}


import java.util.Arrays;

public class GameOfBeans {

    enum Player {JABA, PIETON}

    private final Player firstPlayer;
    private final int depth;
    private byte[] piles;
    private int[][] pietonPrefixCache, pietonSuffixCache;

    public GameOfBeans(Player firstPlayer, int depth, byte[] piles) {
        this.firstPlayer = firstPlayer;
        this.depth = depth;
        this.piles = piles;
    }
```

```java
  public int score() {
// If Pieton plays first, change the game to a subset where JABA plays first, since Pieton is easy to calc first move
      switch (firstPlayer) {
          case JABA:
              break;
          case PIETON:
              byte pietonPlay = pietonPlay(0, piles.length);
              piles = Arrays.copyOfRange(piles, toPrefix(pietonPlay), piles.length - toSuffix(pietonPlay));
              break;
          default:
              throw new IllegalStateException("No firstPlayer");
      }
      pietonPrefixCache = new int[piles.length][depth];
      pietonSuffixCache = new int[piles.length][depth];
      cachePietonPlays();
      //could use unidimensional array ~piles.length * min(piles.length, depth) sized
      //There is a pattern whereby each row has -1 length than the prior, making a stair type pattern
      int rows = Math.min(piles.length + 1, 2 * depth);
      short[][] score = new short[rows][piles.length];
      score[0] = new short[piles.length + 1];
      for (int i = 0; i < piles.length; i++) {
          score[1][i] = piles[i];
      }
      for (int i = 2; i < rows; i++) {
          Arrays.fill(score[i], Short.MIN_VALUE);
      }

      for (int length = 2; length <= piles.length; length++) {
          for (int prefixIndex = 0, suffixIndex = prefixIndex + length; suffixIndex <= piles.length; prefixIndex++, suffixIndex++) {
              int maxPilesToRemove = Math.min(length, depth);
              short maxScore = Short.MIN_VALUE;
              for (int removed = 1, prefixScore = 0, suffixScore = 0; removed <= maxPilesToRemove; removed++) {
                  prefixScore += piles[prefixIndex + (removed - 1)];
                  byte pietonPlay = pietonPlayCached(prefixIndex + removed, suffixIndex);
                  byte pietonPrefix = toPrefix(pietonPlay);
                  int removedPiles = removed + pietonPrefix + toSuffix(pietonPlay);
                  maxScore = (short) Math.max(maxScore,
                          score[(length - removedPiles) % rows][prefixIndex + removed + pietonPrefix] + prefixScore);

                  suffixScore += piles[suffixIndex - removed];
                  pietonPlay = pietonPlayCached(prefixIndex, suffixIndex - removed);
                  pietonPrefix = toPrefix(pietonPlay);
                  removedPiles = removed + pietonPrefix + toSuffix(pietonPlay);
                  maxScore = (short) Math.max(maxScore,
                          score[(length - removedPiles) % rows][prefixIndex + pietonPrefix] + suffixScore);
              }
              score[length % rows][prefixIndex] = maxScore;
          }
      }
      return score[piles.length % rows][0];
  }

  private byte toSuffix(byte pietonPlay) {
      return pietonPlay > 0 ? pietonPlay : 0;
  }
```

```
private byte toPrefix(byte pietonPlay) {
    return pietonPlay < 0 ? (byte) -pietonPlay : 0;
}

private void cachePietonPlays() {
    int maxPilesToRemove = Math.min(piles.length, depth);
    for (int suffixIndex = 0; suffixIndex < maxPilesToRemove; suffixIndex++) {
        cachePietonSuffixPlay(0, suffixIndex);
    }
    for (int prefixIndex = 0, suffixIndex = prefixIndex + maxPilesToRemove; suffixIndex <=
piles.length; prefixIndex++, suffixIndex++) {
        cachePietonPrefixPlay(prefixIndex, suffixIndex);
        cachePietonSuffixPlay(prefixIndex, suffixIndex);
    }
    for (int prefixIndex = piles.length - maxPilesToRemove; prefixIndex <= piles.length;
prefixIndex++) {
        cachePietonPrefixPlay(prefixIndex, piles.length);
    }
}

/**
 * Retrieves pieton plays from cache
 *
 * @param ignoredPrefix ignore before ignoredPrefix (exclusive)
 * @param ignoredSuffix ignore after ignoredSuffix (exclusive)
 * @return positive number if removed from suffix, negative otherwise, 0 if none is removed
 */
private byte pietonPlayCached(int ignoredPrefix, int ignoredSuffix) {
    int pilesRemaining = ignoredSuffix - ignoredPrefix;
    int maxPilesToRemove = Math.min(pilesRemaining, depth);
    if (maxPilesToRemove == 0) {
        return 0;
    }
    int prefixPlay = pietonPrefixCache[ignoredPrefix][maxPilesToRemove - 1];
    int suffixPlay = pietonSuffixCache[ignoredSuffix - 1][maxPilesToRemove - 1];
    return (byte) (toScore(prefixPlay) >= toScore(suffixPlay) ? -prefixPlay : suffixPlay);
}

// Its preferable to spend a little memory on two functions to not have branching if statements
here
   private void cachePietonPrefixPlay(int ignoredPrefix, int ignoredSuffix) {
       int pilesRemaining = ignoredSuffix - ignoredPrefix;
       int maxPilesToRemove = Math.min(pilesRemaining, depth);
       short totalScorePrefix = Short.MIN_VALUE;
       byte maxPrefixRemoved = 0;
       for (short removePrefix = 1, score = 0; removePrefix <= maxPilesToRemove; removePrefix++) {
           score += piles[ignoredPrefix + (removePrefix - 1)];
           if (score > totalScorePrefix) {
               totalScorePrefix = score;
               maxPrefixRemoved = (byte) removePrefix;
           }
           pietonPrefixCache[ignoredPrefix][removePrefix - 1] = toCacheFormat(totalScorePrefix,
maxPrefixRemoved);
       }
   }

   private void cachePietonSuffixPlay(int ignoredPrefix, int ignoredSuffix) {
       int pilesRemaining = ignoredSuffix - ignoredPrefix;
       int maxPilesToRemove = Math.min(pilesRemaining, depth);
       short totalScoreSuffix = Short.MIN_VALUE;
       byte maxSuffixRemoved = 0;
       for (short removeSuffix = 1, score = 0; removeSuffix <= maxPilesToRemove; removeSuffix++) {
           score += piles[ignoredSuffix - removeSuffix];
           if (score > totalScoreSuffix) {
               totalScoreSuffix = score;
               maxSuffixRemoved = (byte) removeSuffix;
           }
           pietonSuffixCache[ignoredSuffix - 1][removeSuffix - 1] = toCacheFormat(totalScoreSuffix,
maxSuffixRemoved);
       }
   }
```

```java
    private short toScore(int pietonCacheEntry) {
        return (short) (pietonCacheEntry >> Short.SIZE);
    }

    /**
     * Encodes the given score on an int with the following pattern: 0xff00
     * and the number of removed piles with the following pattern: 0x000f
     * (independent of prefix vs suffix play)
     * @param score this Pieton play score
     * @param removed this Pieton play removed piles
     * @return encoded Pieton play
     */
    private int toCacheFormat(short score, byte removed) {
        return (score << Short.SIZE) + removed;
    }

    /**
     *
     * @param ignoredPrefix ignore before ignoredPrefix (exclusive)
     * @param ignoredSuffix ignore after ignoredSuffix (exclusive)
     * @return positive number if removed from suffix, negative otherwise, 0 if none is removed
     */
    private byte pietonPlay(int ignoredPrefix, int ignoredSuffix) {
        int prefixPlay = pietonPrefixPlay(ignoredPrefix, ignoredSuffix);
        int suffixPlay = pietonSuffixPlay(ignoredPrefix, ignoredSuffix);
        return (byte) (toScore(prefixPlay) >= toScore(suffixPlay) ? -(byte) prefixPlay :
suffixPlay);
    }

    private int pietonPrefixPlay(int ignoredPrefix, int ignoredSuffix) {
        int pilesRemaining = ignoredSuffix - ignoredPrefix;
        int maxPilesToRemove = Math.min(pilesRemaining, depth);
        short totalScorePrefix = Short.MIN_VALUE;
        byte maxPrefixRemoved = 0;
        for (short removePrefix = 1, score = 0; removePrefix <= maxPilesToRemove; removePrefix++) {
            score += piles[ignoredPrefix + (removePrefix - 1)];
            if (score > totalScorePrefix) {
                totalScorePrefix = score;
                maxPrefixRemoved = (byte) removePrefix;
            }
        }
        return toCacheFormat(totalScorePrefix, maxPrefixRemoved);
    }

    private int pietonSuffixPlay(int ignoredPrefix, int ignoredSuffix) {
        int pilesRemaining = ignoredSuffix - ignoredPrefix;
        int maxPilesToRemove = Math.min(pilesRemaining, depth);
        short totalScoreSuffix = Short.MIN_VALUE;
        byte maxSuffixRemoved = 0;
        for (short removeSuffix = 1, score = 0; removeSuffix <= maxPilesToRemove; removeSuffix++) {
            score += piles[ignoredSuffix - removeSuffix];
            if (score > totalScoreSuffix) {
                totalScoreSuffix = score;
                maxSuffixRemoved = (byte) removeSuffix;
            }
        }
        return toCacheFormat(totalScoreSuffix, maxSuffixRemoved);
    }
}
```