Looking for Bananas in the Monkeys island

by Ernesto De Santis.

Solution

How there is no way to go left, we can assume no cycles, and how there's no valid movement going down, or up, we can assume we can just look to the right column for the next movements.

The solution proposed was based on solving starting from right to left, calculating the smallest paths first. At the first iteration we just get each value as the best path, given that is no more where to go. At the second iteration in each position the best path is the value at the current position plus the best value of one of the three posibles options.

Repeat the algorithm until it reach the first column. Then get the max value calculated at this point.

3x3 Jungle example

Jungle

1	3	3
2	1	4
0	6	4

First Iteration getting values from the last column

-	-	3
-	-	4
-	-	4

Second iteration get the value from second column + best path in the next column alread calculated

-	7	3
-	5	4
-	10	4

Thirst iteration, repeating the algorithm

8	7	3
12	5	4
10	10	4

Finally get the max value among 8, 12, and 10.
Result on this example = 12.

Time execution complexity

Each instruction in positionFrom is O(1), then it's O(1)

```
public static Position positionFrom(Position position, Movement
movement) {
   validation(position);
   validation(movement);

   Position newPosition = new Position(position.getX(),
position.getY());

   if(RIGHT.equals(movement)){
      newPosition.moveRight();
   } else if(RIGHT_UP.equals(movement)){
      newPosition.moveRightUp();
   } else if(RIGHT_DOWN.equals(movement)){
      newPosition.moveRightDown();
   } else {
      throw new JungleException(String.format("Unknow movement
%s", movement));
   }

   return newPosition;
}
```

Each instruction in getQtdWithMovemnt is O(1), then it's O(1)

```
private int getQtdWithMovement(List<Integer> column, Position
position, Movement movement) {
   Position newPosition = Position.positionFrom(position,
movement);

   if(newPosition.getY() >= 0 && newPosition.getY() <
column.size()){
      return column.get(newPosition.getY());
   } else {
      return 0;
   }
}
```

Each instruction in getBestMovemntFrom is O(1), then it’s O(1)

```
private int getBestMovementFrom(List<Integer> column, Position
position) {
   return max(
           getQtdWithMovement(column, position, RIGHT),
               max(
                   getQtdWithMovement(column, position, RIGHT_UP),
                   getQtdWithMovement(column, position, RIGHT_DOWN)
               )
   );
}
```

Each instruction in getBananas is O(1), then it’s O(1)

```
public int getBananas(Position position){
   validateIntoJungle(position);
   int bananas = field[position.getY()][position.getX()];
   if(bananas < 0)
       throw new JungleException(format("A quantity of bananas
cant be negative: %d", bananas));
   return bananas;
}
```

Each instruction in getBestFrom is O(1), then it’s O(1)

```
private int getBestFrom(Jungle jungle, Position position,
List<Integer> column) {
   return jungle.getBananas(position) +
getBestMovementFrom(column, position);
}
```

Look method, the algorithm main’s method has complexity O(x*y) because it has x executions of a block that has y executions. All others instructions have O(1) complexity. Finally is another instruction that iterate over y elements, with O(y). But as O(x*y) > O(y) the final complexity is O(x*y)

```
public int look(Jungle jungle) {
   validate(jungle); //O(1)
   int ySize = jungle.getYJungleSize();//O(1)
   int xSize = jungle.getXJungleSize();//O(1)

   List<Integer> previousColumn =
getInitializedColumn(ySize);//O(y)

   for (int x = xSize-1; x >= 0 ; x--) {//O(x*y)
       List<Integer> currentColumn = new ArrayList<>(xSize);//O(1)
```

```
        for (int y = 0; y < ySize; y++) {//O(y)
            //save banana's quantity best path from this position
            currentColumn.add(getBestFrom(jungle, new Position(x ,
y), previousColumn));//O(1)
        }
        previousColumn = currentColumn;//O(1)
    }

    return previousColumn.stream().reduce(0, Math::max); //O(y)
}
```

Memory complexity

The look method parameter has y*x elements, which is the biggest method memory utilization. But analyzing just the memory allocated by the algorithm we have two arrays of y element at time. Then the memory complexity is O(y)