Group 6:
Sara Savitz
Marilyn Salvatierra
Tiffany Jackson

Project 2 Report

1.) *Choose one of the two problems (exact or bounded size). How you can break down a large problem instance into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the subproblems and why this breakdown makes sense.*

In order to find the count of total states *bounded* by r rows and c columns in the game of Chomp, we must first understand how small problem instances can provide insight into solving larger problem instances. Consider the smallest possible problem instance, a 1 by 1 chocolate bar; there are only two possible states that this bar can have: either (1) the square is eaten, or (2) the square is not eaten. Now consider a 2 by 2 chocolate bar; in this instance, we must consider the dependencies that exist between each square as determined by the rules of Chomp. We know that the upper left square is poison, and the poison will always be the last square eaten. For this reason, we can assign the poison square to be the base case with a state value of 2. We also know that squares to the right and below an eaten square will also be eaten. This implies that the two neighboring squares below or to the right of the poison square will have a single dependency that points to it, either up or to the left. We can assign their state values to be 3, which is 1 plus the state value from our base case. The last square, the bottom right, will in fact have two dependencies, since it is both below and to the right of the poison square. By making this last square dependent on the squares above it and to the left of it, we can calculate the total number of states bounded by a 2 by 2 chocolate bar by summing the state values of both dependencies; this value is then stored in the bottom right square. Since the two squares both have a state value of 3, the total number of states in a 2 by 2 is equal to 6.

As the problem instance grows in size, we can use the cases observed in the smaller problem instance to develop a naive recurrence. This naive recurrence would consist of the four cases observed in the 2 by 2 example: (1) the base case, (2) the right neighbor, (3) the down neighbor, (4) the right and down neighbor. Using this implementation, the state value of the bottom right square of any chocolate bar will always be the total number of possible states that chocolate bar can have.

2.) *What recurrence can you use to model this problem using dynamic programming?*

$$T(r, c) = \begin{cases} 2 & \text{if } r = 0 \text{ and } c = 0 \\ T(r - 1, c) + 1, & \text{if } c = 0 \\ T(r, c - 1) + 1, & \text{if } r = 0 \\ T(r - 1, c) + T(r, c - 1), & \text{otherwise} \end{cases}$$

Recurrence: $T(r,c) = T(r - 1, c) + T(r, c - 1) + \Theta(1)$

Group 6:
Sara Savitz
Marilyn Salvatierra
Tiffany Jackson

3.) *What are the base cases of this recurrence?*

The base case of this recurrence occurs when r = 0 and c = 0, meaning the square is poison. The edge cases of this recurrence occur when either r = 0 or c = 0, meaning the square does not have an 'UP' or 'LEFT' dependency respectively.

4.) *Describe a pseudocode algorithm that uses memoization to compute the solution to your chosen problem.*

Input: r,c - The number of rows and columns for the chocolate bar
Output: The number of valid states that contain r or fewer rows and c or fewer columns
Algorithm: MemoChomp

```
if(chocBar[r][c] > 0)
{
    return chocBar[r][c];
}
if r == 0 and c == 0 then
{
   return 2;
}
else if r == 0 then
{
   chocBar[r][c] = MemoChomp(r, c - 1) + 1;
}
else if c == 0 then
{
   chocBar[r][c] = MemoChomp(r - 1, c) + 1;
 }
 else
 {
   chocBar[r][c] =MemoChomp(r, c - 1) + MemoChomp(r - 1, c);
 }
return chocBar[r][c];
```

Group 6:
Sara Savitz
Marilyn Salvatierra
Tiffany Jackson

5.) *What is the time complexity of your memoized algorithm?*

Total Runtime = # of cells * Cost/Cell + Initialization

$= (r * c) * \theta(1) + \theta(rc)$

$= \theta(rc)$

6.) *Describe an iterative algorithm for the same problem.*

Input: r, c - The number of rows and columns for the chocolate bar
Output: The number of valid states that contain r or fewer rows and c or fewer columns
Algorithm: IterativeChomp

```
for i = 0 to r - 1 do
{
    for j = 0 to c - 1 do
    {

      if i == 0 and j == 0 then
     {
              chocBar[i][j] = 2;
     }
     else if r == 0 then
     {
              chocBar[i][j] = chocBar[i][j - 1] + 1;
     }
      else if c == 0 then
     {
              chocBar[i][j] = chocBar[i - 1][j] + 1;
     }
     else
     {
              chocBar[i][j] = Chomp[i][j - 1] + Chomp[i - 1][j];
      }
    }
  }
return chocBar[r-1][c-1];
```

Group 6:
Sara Savitz
Marilyn Salvatierra
Tiffany Jackson

7.) *Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm? Justify your answer.*

The space complexity of the iterative algorithm can be improved relative to the memoized algorithm by changing the way data is stored in memory. Instead of accessing information from a two-dimensional array each iteration, the algorithm would only access the parts of the data structure it needed. This can be accomplished by declaring 2 one-dimensional arrays that represent the upper row and left column; these arrays will contain the values that serve as our dependencies. Space complexity will be improved for every edge case throughout the iteration, since the edge cases contain only one dependency rather than two, meaning only one array will need to be accessed. As we iterate through the chocolate bar, previously used values are overwritten by new values that will be used as dependencies for later iterations, significantly reducing the number of values needing to be stored. This greatly improves space complexity for very large problem instances.

8.) *Describe an extended version of the memoized algorithm that computes the solution to both problems (exact and bounded size). This algorithm (i.e., the wrapper function) should return an ordered pair containing the two solutions.*

The memoized psuedocode algorithm from question 4 is capable of solving both problems in the same wrapper function. This is because the exact solution is simply a subset of the bounded solution, and can be calculated by passing in r - 1, and c - 1 to the memoized function. Ideally, the solution to the bounded problem would be invoked first, and the exact solution invoked second; this makes it so the second function call would only have to read the results stored in the data structure calculated from the first function call.

```
wrapper(r, c)
{
        chocBar = Array(r,c);
        Initialize chocBar values to 0;
        totalStates = Chomp(r, c);
        exactStates = Chomp(r - 1, c - 1);
        return (exactStates, totalStates);
}
```