Alan Rodriguez
Chris Salgado
Austin Erickson
COT 4400
Project 2

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.

The bounded size problem consists of counting the number of valid game states that have r or fewer rows and c or fewer columns. Since no valid game state can have more than r rows or c columns, we can say that all all valid game states have r or fewer rows and c or fewer columns. Thus, we are actually finding the number of valid game states that exist for a particular game.

We could start counting our valid game states from the end state (having no tiles left), and work our way backwards, adding tiles to the game instead of taking them away. The first tile to add would be the poison tile, creating a game of one row and one column. We know that there are solely two states that can occur if there is only one tile: either it is there or it isn't. So the minimum possible number of valid states we could have for a 1x1 game is two, and we can build up to the board we are presented with by adding tiles to this base case one at a time. If we add a tile to either the side or below the first tile we have added only one extra possible state, thus as we add tiles in a single row or single column away from the poisoned tile, the total number of states is equal to the number of tiles on the board plus one. When we start adding additional tiles that are not in the first row or the first column, we add more than one possible valid state to the game. The number of valid states on a rectangular board is the sum of the number of valid states of the board one row smaller, with the number of valid states of the board one column smaller. By following these rules as we place tiles on the game board, we can know how many possible valid states there will be for any game board of r rows and c columns.

By following the above method of adding tiles to the game board, we can implement an iterative algorithm to go through each index in the 2d array, starting from the top left and working our way down and to the right. A recurrence can also be used to start from any index in the array, and calculate the number of valid game states by recursing up and left, and is discussed below.

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

T(n) = { Θ(1) if i = 0 and j = 0 otherwise,
  2T(i+j) +Θ(n) if i = 0 or j = 0 else,
  2T(i*j) +Θ(n) }

The recurrence is based on dependencies pointing up and left.The middle is left as + since either i or j is 0 it'll only access one loop so essentially its Θ(n) this case i.

3. What are the base cases of this recurrence?

The base case of this recurrence accounts for two state possibilities, hence the assignment of states[0][0] = 2; These cases are when there is only 1 square, or 0 squares.

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

```
long long memoHelper(int r, int c)
{
	buildArray(r, c);	Θ(r*c)
	initialize(r, c);	Θ(r*c)
	return memoChomp(r-1,c-1);
}

long long memoChomp(int i, int j)
{
	if(states[i][j] > 0)
		return states[i][j];

	if(i == 0 && j == 0)
		states[i][j] = 2;
	else
	{
		if(i == 0)
			states[i][j] = 1 + memoChomp(i,j-1);
		else
		{
			if(j == 0)
				states[i][j] = 1 + memoChomp(i-1,j);
			else
				states[i][j] = memoChomp(i-1,j) + memoChomp(i,j-1);
		}
	}
	return states[i][j];
}
long long memoChomps = memoHelper(rows,cols);
```

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

memoHelper => Θ(r*c)

memoChomp => Θ(r*c) + Θ(1)

6. Describe an iterative algorithm for the same problem.

```
long long iterativeChomp(int r, int c)
{
	for(int i=0; i<r; i++)
		for(int j=0; j<c; j++)
		{
			if(i == 0 && j == 0)
				states[i][j] = 2;
			else
			{
				if(i == 0)
					states[i][j] = 1 + states[i][j-1];
				else
				{
					if(j == 0)
						states[i][j] = 1 + states[i-1][j];
					else
						states[i][j] = states[i-1][j] + states[i][j-1];
				}
			}
		}
	return states[r-1][c-1];
}
```

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

Yes, basing dependencies of current cell in array to point to elements in the above row reduces space complexity to linear time of O(k) where k is the size of the above row. In order to do this, the new algorithm would involve only storing dependencies needed for the current row.

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.

An exact solution is only a subset of a bounded solution. Notice a 2 by 2 has 6 valid possible states, a 3 by 3 has 20 valid possible states, and 6 exact possible states. This implies that the problem can be broken down into sub sets. So a 4 by 4 has 20 exact possible states and a 4 by 4's total valid states would be equivalent to a 5 by 5's exact possible states because it would involve the same amount of possible combinations with the same dimensions of rows and columns when you consider an exact solution set is a total valid solution set of 1 row and 1 column less.

With a memoized algorithm, all the values are stored into the array. The memoized check will check if the array element already has a value assigned to it other than its initial. If so, it will return the value stored and it will not recalculate.

```
long long* memoPair = memoHelper(rows,cols);
long long* memoHelper(int r, int c)
{
	long long* Set {};
	States** = buildArray(r, c);
	initialize(r, c);
	Set[0] = return memoChomp(r-1,c-1);
	Set[1] = return memoChomp(r,c);

	Return Set;
}
long long memoChomp(int i, int j)
{
	if(states[i][j] > 0)
		return states[i][j];

	if(i == 0 && j == 0)
		states[i][j] = 2;
	else
	{
		if(i == 0)
			states[i][j] = 1 + memoChomp(i,j-1);
		else {
			if(j == 0)
				states[i][j] = 1 + memoChomp(i-1,j);
			else
				states[i][j] = memoChomp(i-1,j) + memoChomp(i,j-1);
		}
	}
	return states[i][j];
}
```