Project 2: Chomp Game States

COT4400: Analysis of Algorithms

Henry Cardenas

Grant Gurvis

David Hatcher

1.

To solve the problem of the number of current game states given a number of squares, n, a maximum number of rows, r, and a maximum number of columns, c, it should be broken down into smaller subproblems. The most efficient way to break this down would be to find the number of valid game states for n, r, and c specifically, meaning that all columns and all rows have at least a single square in them. Once you have this value for the number of game states at a specific size you can add the game states for smaller sizes with the same number of pieces as well. The values to add would be the valid game states for the inputs n, r-1, c and n, r, c-1. However, these values will contain overlapping cases as the valid game states at n, r-1, c will include the valid game states of n, r-2, c, and n, r-1, c-1, while the valid game states at n, r, c-1 will include the valid game states at n, r-1, c-1 and n, r, c-2. Therefore, we must remove the overlapping game states between these two subproblems.

Thus, that any problem with arbitrary inputs n, r, and c can be solved by finding the current number of game states for that specific size, as well as including the game states for the new subproblems with inputs of n, r-1, c and n, r, c-1 and removing the overlapping game states from n, r-1, c-1.

2.

$$Chomp(n,r,c) = Chomp(n,r-1,c) + Chomp(n,r,c-1) - Chomp(n,r-1,c-1) + \#\ of\ game\ states\ for\ n\ pieces\ in\ r*c\ board$$

3.

Base Cases:

i. 1, if r*c = n
ii. 0, if r = 1 and c > n
iii. 0, if c = 1 and r > n

4.

```
Input: n, number of squares

Input: r, number of rows

Input: c, number of columns

Algorithm chomp


chomp(n,r,c)

1    chompArr = array[r,c];

2    init chompArr to -1;

3    return MemoChomp(n,r,c);


MemoChomp(n,r,c):

1    if chompArr[r,c] == -1 then

2        if r == 1 and n > c or c == 1 and n > r then

3            chompArr[r,c] = 0;

4        else if r*c == n then

5            chompArr[r,c] = 1;

6        else

7            chompArr[r,c] = MemoChomp(n,r-1,c) + MemoChomp(n,r,c-1) - MemoChomp(n,r-
1,c-1) + # of current game states;

8        end

9    end

10   return chompArr[r,c];
```

5.

The initial chomp algorithm will have be O(r*c) as this is the initialization cost and all other operations within that function are O(1).

For the getChompGS algorithm, the algorithm will need to fill r*c slots in the worst case. As each of these slots will take a total O(1) to fill once the first base case is hit, it will come out to a total complexity of O(r*c) for the entirety of the algorithm.

6.

```
Input: n, number of squares

Input: r, number of rows

Input: c, number of columns

chomp(n,r,c)

1    chompGS = array[r,c];

2    init chompArr to -1;

3    return IterChomp(n,r,c);


IterChomp(n,r,c):

1    for i = 1 to r step = 1 do

2        for j = 1 to c step =  1 do

3            if i == 1 and n > j or j == 1 and n > i do

4                chompArr[i,j] = 0;

5            else if i*j == n do

6                chompArr[i,j] = 1;

7            else

8                chompArr[i,j] = chompArr[i-1,j] + chompArr[i,j-1] - chompArr[i-1,j-1] +
# of current game states for ixj size;

9            end if

10       end for

11   end for

12   return chompArr[r,c];
```

7.

The space complexity of this iterative algorithm could be improved relative to the memoized algorithm. As this algorithm iterates from the top left of the matrix to the bottom right it allows the algorithm to be able to optimize the space complexity by giving the ability to store just the current row and previous row. This is accomplishable because the only values needed to calculate the current values are the value one space to the left and one space up.