# TCSS 343 Programming Assignment

By: Brandon Watt and Andrew Gates

## 3.1 Brute Force -

Our brute force method uses the recursiveHelperBF to traverse the first line of the array from left to right, initially setting the minimum value and route to be the value at index table[0][table.length] thus making only table.length - 2 calls to the recursiveBF method. The recursiveHelperBF method also manages the route arrays by assigning the initial indices that are known, reassigning the arrays if a new min is a found, and also resetting the arrays to 0 for the next iteration of the for loop.

Our recursiveBF method has an if statement that will check to see if the current index is out of bounds. If it is it will check if the current sum is less than the min, which if that is true it will reassigning the min and the route of the min to be the sum and the route of the sum. The corresponding else statement will update the route for the current index, and then make Current Row Length - 2 calls updating the sum during each of those calls.

Back in recursiveHelperBF the route will be updated if the sum that is returned from the initial recursiveBF call is less than the current min. In which case the route will be updated as well to this new route. Once the for loop is complete it will reverse the route to make the route into increasing order, then print the min and the route that corresponds to it.

The complexity of the recursiveHelperBF method is -

- $\Theta(1)$ for the declarations.
- $\Theta(n-2) \in \Theta(n)$ for the outer for loop.
    - $\Theta(n)$ for the inner for loop that resets the route values to 0
    - $\Theta(1)$ for the declarations.
    - $\Theta(1)$ for the if statement.
        - $\Theta(n)$ for the for loop inside the if that copies tempRoute2 onto route.
- $\Theta(n)$ for the reversal of route.
- $\Theta(1)$ for the final prints.

The complexity of the recursiveBF method is -

- $\Theta(n)$ for the recursiveHelperBF calls.
    - $\Theta(1)$ for the declarations.
    - $\Theta(1)$ for the if statement checking to see if the current index is out of bounds.
        - $\Theta(1)$ for the if statement checking to see if the current sum is less than min.
            - $\Theta(1)$ for the declarations.
            - $\Theta(n)$ for for loop inside the if that copies tempRoute onto tempRoute2.
    - $\Theta(1)$ for else statement.

        - $\sum_{n=1}^{x}$ (n) : where x is the Current Row Length results in - $(1/2)x(x+1) = (x^2)/2 + x/2 = \Theta(x^2)$ where x is $\leq$ n - 2 = $\Theta(n^2)$
        - $\Theta$ (1) for the declarations.
    - $\Theta$ (1) for return.

The final complexity comes from the $\Theta(n)$ outer loop calls in the recursiveHelperBF method multiplied with the $\Theta(n^2)$ of recursive calls in the recursiveBF method which results in a final complexity of $\Theta(n^3)$ .

## 3.2 Divide and Conquer -

Our divide and conquer method uses the recursiveHelperDAC to create the array and tempArray and fills them with Integer.MAX_VALUE and N respectively. Then it makes the initial call to recursiveDAC and after that call is done it will print the min and the route of that min.

Our recursiveDAC method has an if statement to check to make sure the current index is within the bounds of our table. If it is it will add that index to the tempRoute, and then make two recursive calls and calculate the min between the two. One of the two calls will be to add the current index to the sum and go to the next index then increment the position, while the other call will be to keep the current sum and then go to the next index but don't increment the position. The corresponding else statement will add the current index to the sum and update the tempRoute. Then it will check if the current sum is less than our min and if it is it will reassign min to be that value, as well as copying tempRoute onto our min route.

The complexity of the recursiveHelperDAC method is -

- $\Theta$ (1) for the declarations and prints.
- $\Theta(n)$ for the filling of the route array with Integer.MAX_VALUE.
- $\Theta(n)$ for the filling of the route array with N.

The complexity of the recursiveDAC method is -

- $\Theta(1)$ for the if statement checking to make sure current index is not out of bounds.
    - $\Theta(1)$ for the declarations.
    - $\Theta(nlogn)$ for the two recursive calls made.
- $\Theta(1)$ for the else statement.
    - $\Theta(1)$ for the declarations.
    - $\Theta(1)$ for the if statement checking to see if the current sum is less than the min.
        - $\Theta(1)$ for the declarations.
        - $\Theta(1)$ for the if loop checking to see if the stored min in tempRoute[table.length] is less than the stored min in route[table.length].
            - $\Theta(n)$ for the for loop reassigning the tempRoute values to route.

- $\Theta(1)$ for the return.

The final complexity comes from the $\Theta(nlogn)$ recursive calls. Everything else is an addition of complexity and not nested complexities so the final complexity is $\Theta(nlogn)$.

## 3.3 Dynamic Programming -

Our dynamic programming algorithm creates a priority queue to hold the values of the paths found. It then makes a value, least, that will be used to track the row of the cheapest path. The object that makes it dynamic is the 2D array used to store the path values. The first row of this array is filled with the same values as the given table. The method then enters a double for loop that walks through all the positions in the table a single time.

At each position the array uses a for loop to determine the value of the cheapest path used to get to that location. This value is then stored in the 2D array and used to determine the cheapest path for later values. At the end of each row the value of that path is stored in the priority queue and if that value is the smallest the value least is set to that row. When the loop has finished the algorithm will reconstruct the path using the least value and the values stored in the 2D array. Since this array is built backwards it is reworked by a second loop. The value of the cheapest path as well as its route are then printed to the console.

The complexity of the dynamic method is -

- $\Theta(1)$ for the declarations.
- $\Theta(n)$ for the for loop to copy the first row of the table onto our summations array.
- $\Theta(n-2) \in \Theta(n)$ for the for loop to traverse the rows of the table.
    - $\Theta(1)$ for the declarations.
    - $\Theta(n)$ for the loop to traverse the columns of the table.
        - $\Theta(1)$ for the declarations.
        - $\Theta(n)$ for the for loop to find the smallest value.
    - $\Theta(nlogn)$ for adding to the heap.
- $\Theta(1)$ for the declarations.
- $\Theta(n-2) \in \Theta(n)$ for the for loop to construct the initial route array.
    - $\Theta(1)$ for the declarations and updating of route and least.
    - $\Theta(n)$ for the for loop to reassign min and position.
        - $\Theta(1)$ for the declarations and if statement to determine positions.
- $\Theta(n)$ for the for loop to make a new route array without the extra zeroes.
    - $\Theta(1)$ for the if and else statements to eliminate the zeroes and copy to finished.
- $\Theta(1)$ for the printing of min and the route associated with min.

The final complexity comes from the traversal for loops, which is $\Theta(n)$ for the rows, $\Theta(n)$ for the columns and $\Theta(n)$ to find the smallest value. Resulting in a final complexity of $\Theta(n^3)$

## 4.1 Dynamic Programming -

The algorithm we designed will create a s*s box inside of the n*n array with the target position (x,y) being the top left index of our s*s box. Then we will scan through our s*s box traversing it linearly scanning for a TRUE value. If there is no TRUE values then we will return TRUE as the answer.

If there is a TRUE value then we will move our s*s box one index to the left.

We will repeat this left traversal scanning for TRUE values and repeat the shifting of our s*s box until it gets shifted s-1 amount of times in which case we will reset the s*s box so that the box moves upwards in the y-axis now repeating similarly to the x-axis traversal.

We will repeat this left traversal scanning for TRUE values and repeat the shifting of our s*s box until it gets shifted s-1 amount of times in which case we will reset the s*s box so that the box now moves in a diagonal direction up and to the left. Repeating similarly to the x-axis and y-axis traversals.

Once the s*s box is shifting s-1 amount of times it will have checked all possible combinations and will have either returned TRUE that a s*s box can be found, or it will return FALSE if it reaches the end and hasn't found a box yet.

```
FieldAndStones(boolean field[n][n], int targetPosition, int s)
{
        Let result be a boolean value to be updated if there are TRUE values found, initialized to TRUE;
        Let xShift be an int value to control the shifting of the s*s array in the x-axis.
        Let yShift be an int value to control the shifting of the s*s array in the y-axis.
        Let xyShift be an int value to control the shifting of the s*s array in the diagonal direction.

        if(x,y == TRUE)
        {
                return FALSE;
        }

        for(int i = x; i < x + s; i++)
        {
                for(int j = y; j < y + s; j++)
                {
                        if(field[i][j] == FALSE)
                        {
                        }
                        else
                        {
                                result == FALSE;
```

```
                }
        }
}

if(result == TRUE)
{
        return TRUE;
}
result == TRUE;

for(xShift = 1; xShift < s; xShift++)
{
        for(int i = x - xShift; i < x - xShift + s; i++)
        {
                for(int j = y; j < y + s; j++)
                {
                        if(field[i][j] == FALSE)
                        {
                        }
                        else
                        {
                                result == FALSE;
                        }
                }
        }
        if(result == TRUE)
        {
                return TRUE;
        }
        result == TRUE;
}

for(yShift = 1; yShift < s; yShift++)
{
        for(int i = x; i < x + s; i++)
        {
                for(int j = y - yShift; j < y - yShift + s; j++)
                {
                        if(field[i][j] == FALSE)
                        {
                        }
                        else
                        {
```

```
                                        result == FALSE;
                                }
                        }
                }
                if(result == TRUE)
                {
                        return TRUE;
                }
                result == TRUE;
        }

        for(xyShift = 1; xyShift < s; xyShift++)
        {
                for(int i = x - xyShift; i < x - xyShift + s; i++)
                {
                        for(int j = y - xyShift; j < y - xyShift + s; j++)
                        {
                                if(field[i][j] == FALSE)
                                {
                                }
                                else
                                {
                                        result == FALSE;
                                }
                        }
                }
                if(result == TRUE)
                {
                        return TRUE;
                }
                result == TRUE;
        }
        return FALSE;
}
```

The complexity of this algorithm is $\Theta(n^3)$ since the triple for loops result in a complexity of n*n*n which is $\Theta(n^3)$ there are multiple for loops but they are added to each other and not multiplied so it stays at the complexity $\Theta(n^3)$.

Homework 4
By: Brandon Watt and Andrew Gates

## BRUTE FORCE -

**SPACE -**

recursiveHelperBF -
(9* 4 bytes) Integer Values
(4 * 4 bytes * n) 1D Integer Arrays
(4 bytes * n^2) 2D Integer Arrays

recusiveBF -
(7 * 4 bytes * n^2) Integer Values
(2 * 4 bytes * n^2) 1D Integer Arrays
(4 bytes * n^2 * n^2) 2D Integer Arrays

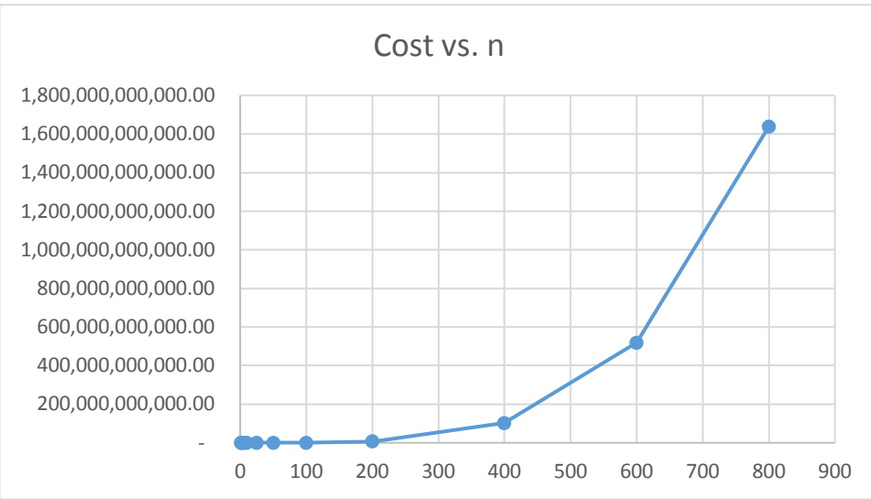
**TIME -**

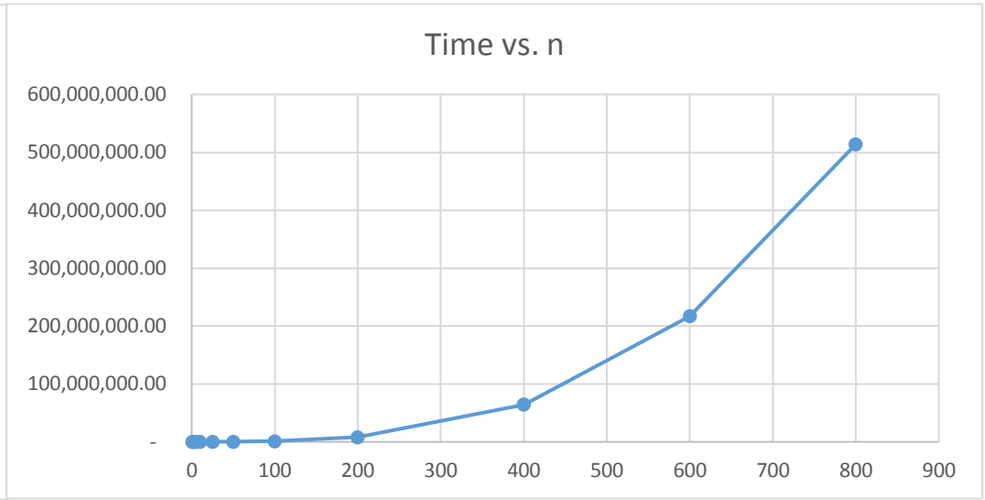
recursiveHelperBF -
(n^2) for resetting routes to 0
(n^2) for copying to route
(n) for reversal of route

recusiveBF -
(n^3) for recursiveHelperBF and recursiveBF calls
(n^2) for copying to tempRoute2

| Cost (Bytes) | n Value |
|---|---|
| 96.00 | 1 |
| 292.00 | 2 |
| 3,616.00 | 5 |
| 44,196.00 | 10 |
| 1,587,936.00 | 25 |
| 25,100,836.00 | 50 |
| 400,401,636.00 | 100 |
| 6,401,603,236.00 | 200 |
| 102,406,406,436.00 | 400 |
| 518,414,409,636.00 | 600 |
| 1,638,425,612,836.00 | 800 |

| Time (Seconds) | n Value |
|---|---|
| 5.00 | 1 |
| 22.00 | 2 |
| 205.00 | 5 |
| 1,310.00 | 10 |
| 17,525.00 | 25 |
| 132,550.00 | 50 |
| 1,030,100.00 | 100 |
| 8,120,200.00 | 200 |
| 64,480,400.00 | 400 |
| 217,080,600.00 | 600 |
| 513,920,800.00 | 800 |




## DIVIDE AND CONQUER -

**SPACE -**

recursiveHelperDAC -
(4 bytes) Integer Values
(2 * 4 bytes * n) 1D Integer Arrays
(4 bytes * n^2) 2D Integer Array

recursiveDAC -
(6 * 4 bytes * nlogn) Integer Values
(2 * 4 bytes * n * nlogn) 1D Integer Arrays
(4 bytes * n^2 * nlogn) 2D Integer Arrays

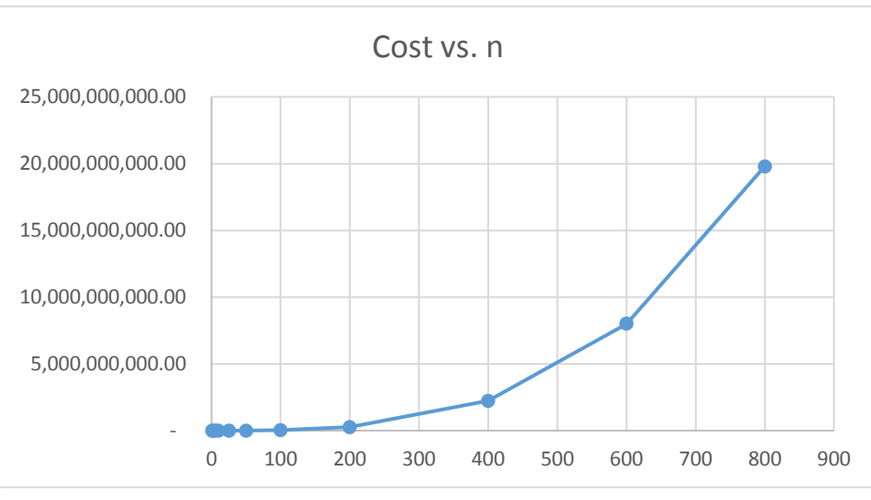
**TIME -**

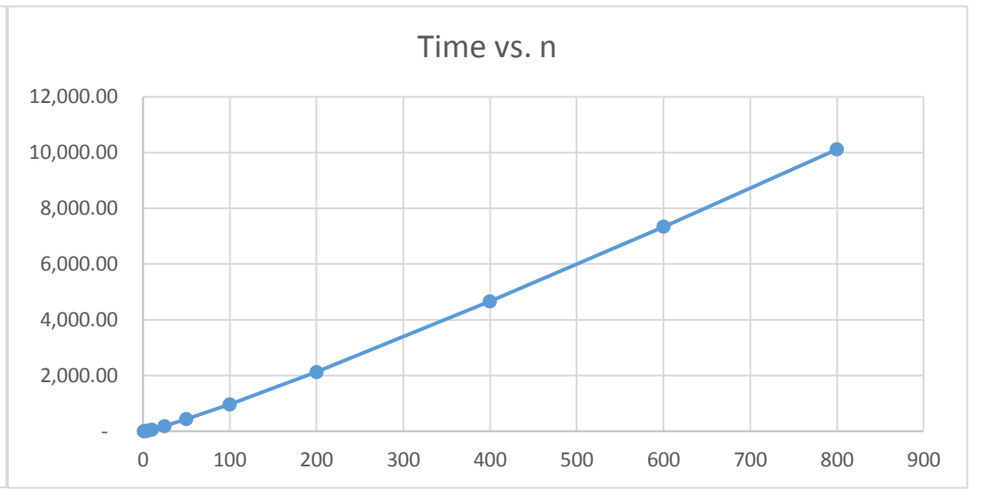
recursiveHelperDAC -
(n) for filling of route
(n) for filling of tempRoute

recursiveDAC -
(nlogn) for recursiveDAC calls
(n) for copying to route

| Cost (Bytes) | n Value |
|---|---|
| 24.00 | 1 |
| 164.00 | 2 |
| 2,087.98 | 5 |
| 17,306.52 | 10 |
| 319,150.61 | 25 |
| 2,952,381.85 | 50 |
| 27,164,482.51 | 100 |
| 247,249,326.56 | 200 |
| 2,224,620,705.52 | 400 |
| 8,001,860,845.41 | 600 |
| 19,802,751,986.39 | 800 |

| Time (Seconds) | n Value |
|---|---|
| 3.00 | 1 |
| 8.00 | 2 |
| 26.61 | 5 |
| 63.22 | 10 |
| 191.10 | 25 |
| 432.19 | 50 |
| 964.39 | 100 |
| 2,128.77 | 200 |
| 4,657.54 | 400 |
| 7,337.29 | 600 |
| 10,115.08 | 800 |



## DYNAMIC PROGRAMMING -

| SPACE - |
|---|
| dynamic - |
| (12 * 4 bytes) Integer Values |
| (2 * 4 bytes * n) 1D Integer Arrays |
| (2 * 4 bytes * n^2) 2D Integer Arrays |
| (4 bytes * n) Priority Queue |

| TIME - |
|---|
| dynamic - |
| (n) for the loop to copy the first row |
| (n^3) for the loops to traverse the table |
| (n^2*logn) for adding to the heap in the loop |
| (n^2 )for the loops to construct the route array |
| (n) for the loop to make a new route array with no zeroes |

| Cost (Bytes) | n Value |
|---|---|
| 68.00 | 1 |
| 104.00 | 2 |
| 308.00 | 5 |
| 968.00 | 10 |
| 5,348.00 | 25 |
| 20,648.00 | 50 |
| 81,248.00 | 100 |
| 322,448.00 | 200 |
| 1,284,848.00 | 400 |
| 2,887,248.00 | 600 |
| 5,129,648.00 | 800 |

| Time (Seconds) | n Value |
|---|---|
| 4.00 | 1 |
| 20.00 | 2 |
| 218.05 | 5 |
| 1,452.19 | 10 |
| 19,202.41 | 25 |
| 141,709.64 | 50 |
| 1,076,638.56 | 100 |
| 8,346,154.25 | 200 |
| 65,543,816.99 | 400 |
| 219,683,574.73 | 600 |
| 518,813,667.96 | 800 |