CITS2200 Data Structures and Algorithms

Final-project Report

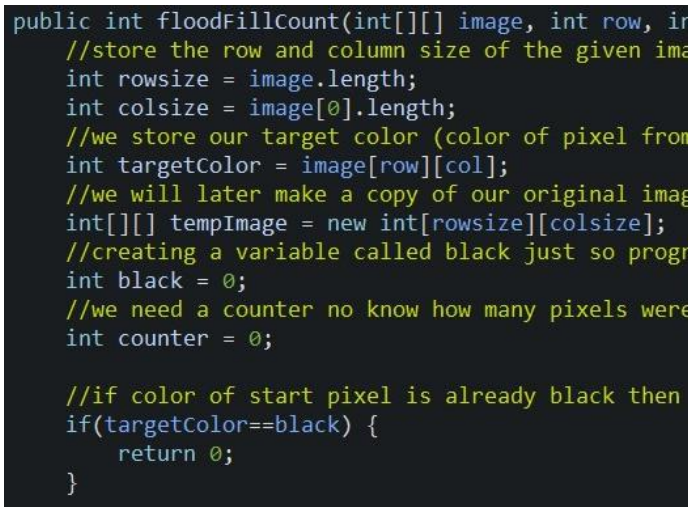
Student – GAURAV CHAKRAVERTY (22750993)

Date – June 5, 2020

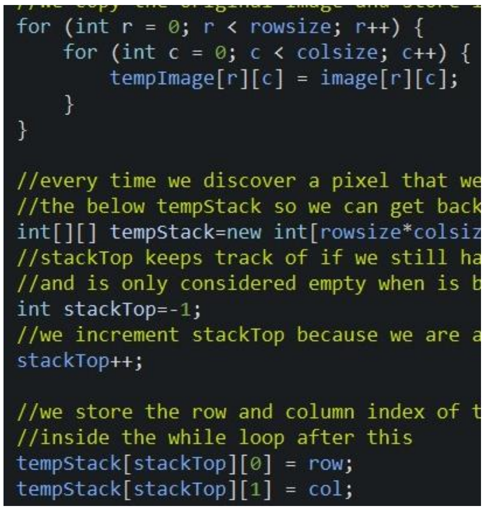
Index

Flood-Fill-Count Explanation

- For the flood fill problem, we were are given a 2D int array as input and then the row and column index of the pixel from where to begin the flood fill operation. The flood fill operation will only color required pixels black and then return the number of pixels that had to be flood filled.
- The way I am going to solve this problem is by checking each pixel adjacent to the starting pixel and then if the adjacent pixel needs to be flood filled as well then, I will process that pixel. I will also have to keep a record of which pixel I have already seen or flood filled so I only have to process each pixel once.

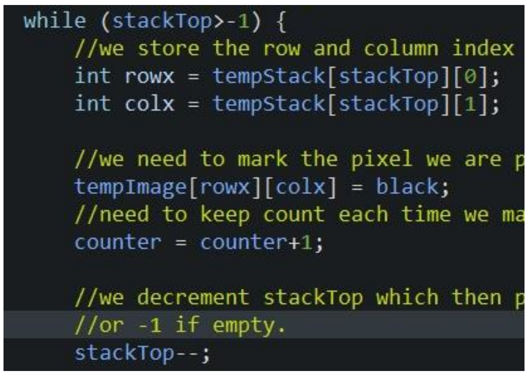


```
public int floodFillCount(int[][] image, int row, ir
    //store the row and column size of the given ima
    int rowsize = image.length;
    int colsize = image[0].length;
    //we store our target color (color of pixel from
    int targetColor = image[row][col];
    //we will later make a copy of our original imag
    int[][] tempImage = new int[rowsize][colsize];
    //creating a variable called black just so progr
    int black = 0;
    //we need a counter no know how many pixels were
    int counter = 0;

    //if color of start pixel is already black then
    if(targetColor==black) {
        return 0;
    }
```



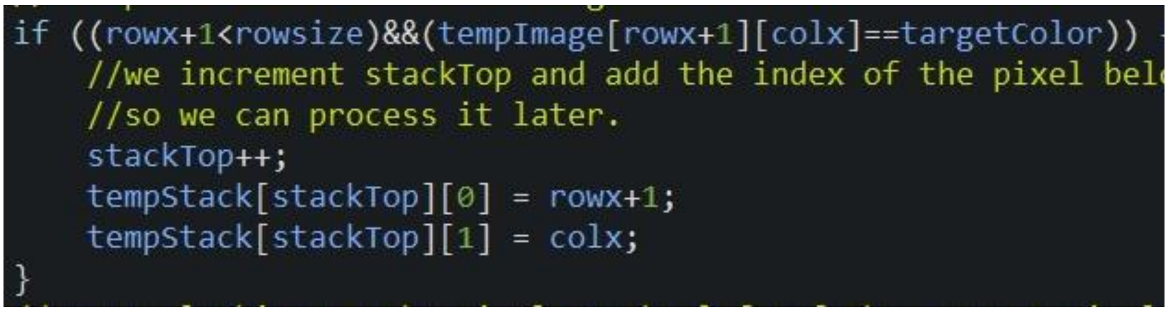
```
//we copy the original image and store i
for (int r = 0; r < rowsize; r++) {
    for (int c = 0; c < colsize; c++) {
        tempImage[r][c] = image[r][c];
    }
}

//every time we discover a pixel that we
//the below tempStack so we can get back
int[][] tempStack=new int[rowsize*colsiz
//stackTop keeps track of if we still ha
//and is only considered empty when is b
int stackTop=-1;
//we increment stackTop because we are a
stackTop++;

//we store the row and column index of t
//inside the while loop after this
tempStack[stackTop][0] = row;
tempStack[stackTop][1] = col;
```

- At the start of my program I check if the target pixel is already black – if so then we do not need to do any flood fill and can return count as 0.
- Next I have created a copy of the input image called tempImage. I will be using tempImage to keep count of which pixels I have already marked black.
- I also make a stack called stackTop using a 2D array and stackTop points to the top of this stack. This stack will be used to store the coordinates of any new pixel that we encounter during our search and determine needs to be floodfilled. We will see how that happens after in the next steps.
- Also, the maximum number of coordinates that we will need to store inside our tempStack is equal to the number of pixels in the image. (If we need to floodfill the whole image).
- Our tempStack is considered empty when stackTop is -1.
- We will now construct a while loop that keeps running until stackTop is empty.
- Before the while loop begins, we store the coordinates of our starting pixel inside our tempStack and increment stackTop. After that we enter the while loop.



```
while (stackTop>-1) {
    //we store the row and column index
    int rowx = tempStack[stackTop][0];
    int colx = tempStack[stackTop][1];

    //we need to mark the pixel we are p
    tempImage[rowx][colx] = black;
    //need to keep count each time we ma
    counter = counter+1;

    //we decrement stackTop which then p
    //or -1 if empty.
    stackTop--;
```

- While the while-loop is running we store the coordinates of the pixel at the top of the stack inside two variables rowx and colx.
- We then color the corresponding pixel in our tempImage as black. We also increment our counter each time we color a pixel black.
- Then we decrement our stackTop value to say that we have dealt with the topmost item in the stack. If we reach a point where stackTop is -1 and the while loop will exit.



```
if ((rowx+1<rowsize)&&(tempImage[rowx+1][colx]==targetColor))
    //we increment stackTop and add the index of the pixel bel
    //so we can process it later.
    stackTop++;
    tempStack[stackTop][0] = rowx+1;
    tempStack[stackTop][1] = colx;
}
```

- I then have 4 if-conditions – each if-condition is very similar and the only difference is that they each check the pixel adjacent to our current pixel in a specific direction. In the picture above, the if-condition checks if the pixel below the current pixel needs to be floodfilled. We do not execute the if-condition if the pixel below is already black or is a color that we do not need to floodfill.
- If we do detect that a specific pixel needs to be floodfilled then we add the coordinates of that pixel inside our tempStack and increment stackTop by 1.
- Each time we find a pixel that need to be floodfilled we keep adding it to our tempStack and each time our while-loop executes we take out the coordinates of the pixel at the top of our tempStack and process it until all stackTop is -1.
- Once stackTop is -1 we exit the while loop and return the value of count or the number of pixels that we have floodfilled.

<u>**Complexity Analysis**</u>

- We take a look at each pixel in the image while making tempImage.
- Besides that, we process each pixel only once and only if it needs to be floodfilled.
- O(2P) where P is the number of pixels can also be written as **O(P).**
- An image with P pixels requires memory of O(P). Then tempImage and tempStack will also require memory of O(P) each.
- Therefore, the total amount of memory we will require is O(3P).

Brightest-Square Explanation

- In the brightest square problem, we need to find out the maximum total brightness of any possible k by k square that fits in the given image.
- The challenge was to think of way with which I do not need to go through each and every row and column every time I was calculating the total brightness of each possible position of the k by k square.
- After searching the internet for a while, I came across the following page on Wikipedia – https://en.wikipedia.org/wiki/Summed-area_table
- The title of the article is "Summed-area table" and talks about quite an interesting concept. I am going to attempt to explain the concept with a few pictures I drew on Excel.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

- Imagine we have the table on the left. Now, we try to make another table of the exact same size as the original but in this table each value is a sum of all the values on its left and top. (including itself). The following pictures will make what I mean more clear -

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

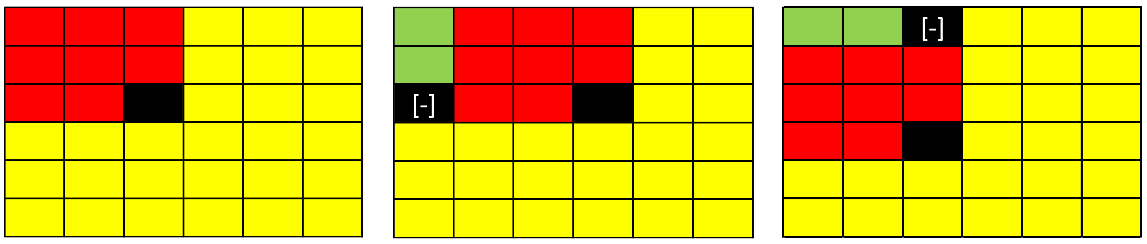
31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

- We can think of the tables as the image provided in our project.
- We managed to get 37 in the picture on the right by adding all the values on the left of the corresponding pixel (including itself) on the left picture. (31+2+4 = 37) There are no values above our selected pixel this point.

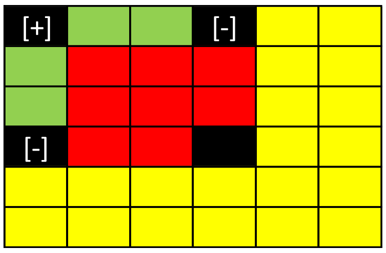
31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

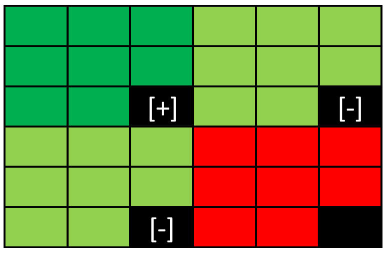
31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

- Here we can see that we get 84 of the image on the right by looking at 9 (which is the corresponding pixel on the image on the left) then we add to 9 all the pixels above and on the left of it.
- But something we can see here is we do not actually need to go through each and every pixel every time. Since we know that 37 is already the sum of all the pixels on the left of it we can just do 37 + 12 + 26 + 9 = 87. (37 is the same as 31 + 2 + 4).
- Hence, at any given point I can just look at the pixels on left of the given pixel from the image on the left and add the pixel above it from the image on the right. Then we will get the pixel which is a sum of all values on its left and above it.
- Therefore, after we have created our summed-area table (image on the right) – we can use it to find the total brightness of our k by k square without going through each and every pixel each time the square shifts.

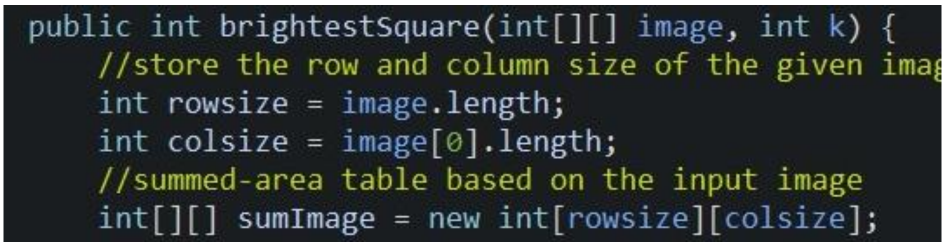


- I have drawn the above pictures to explain how we will calculate the total brightest of our square each time it moves. So imagine k is 3 so we have a 3 by 3 square. In the beginning, the square will be on the top left of our image and the black pixel corresponding to the summed area table will give us the total brightness of the whole square.
- Next when we move the square to the right, we just need to take the value of the black pixel from our summed area table and from that we need to minus the black pixel on the left as we need to exclude the area marked in green.
- We can use the same logic when the square moves down.
- Next when the square has rows and columns above and on the left, we can apply the same logic as before and take the black pixel from the bottom right and then subtract from it the black pixels on the top right and bottom left outside it. Looking at the picture I drew makes it clear to understand. We must also remember to add the value of the black pixel on the top left. This is because otherwise we will end up subtracting the area marked in dark green twice.



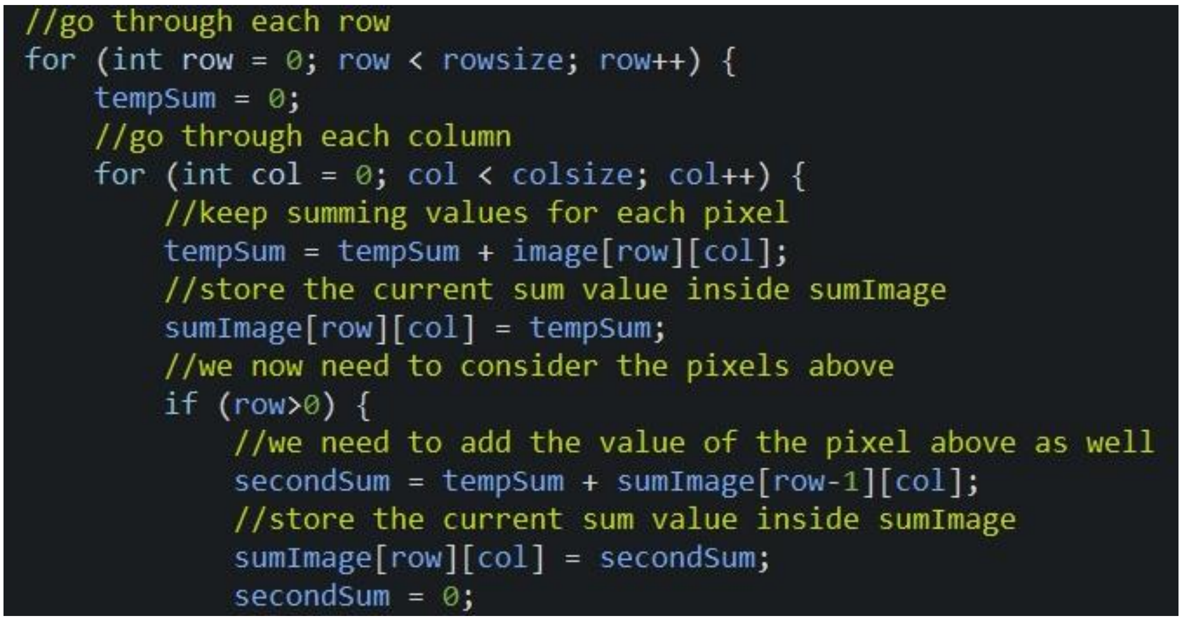


- Let us now look at my program to see I am implemented this concept.



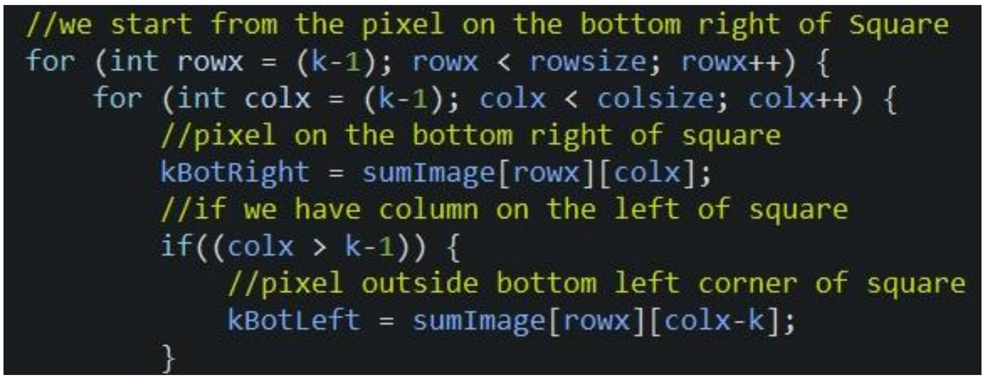
```
public int brightestSquare(int[][] image, int k) {
    //store the row and column size of the given imag
    int rowsize = image.length;
    int colsize = image[0].length;
    //summed-area table based on the input image
    int[][] sumImage = new int[rowsize][colsize];
```

- First, I created a 2D array called sumImage which is the same size as our input image but we will treat this as our summed-area table as we discussed before.



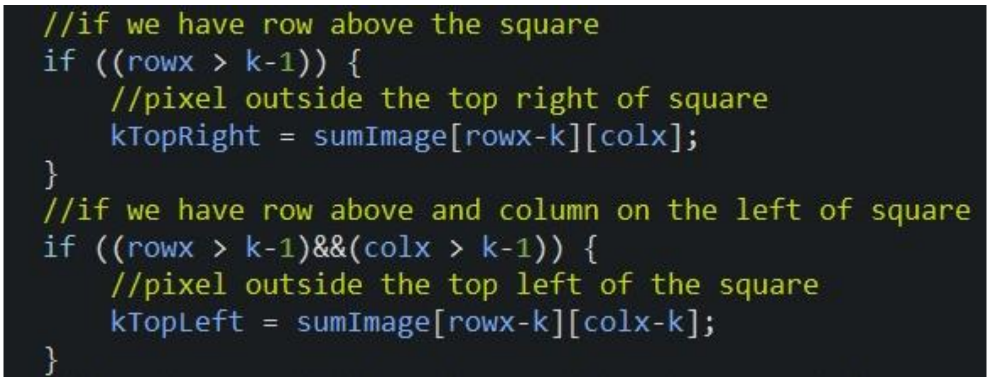
```
//go through each row
for (int row = 0; row < rowsize; row++) {
    tempSum = 0;
    //go through each column
    for (int col = 0; col < colsize; col++) {
        //keep summing values for each pixel
        tempSum = tempSum + image[row][col];
        //store the current sum value inside sumImage
        sumImage[row][col] = tempSum;
        //we now need to consider the pixels above
        if (row>0) {
            //we need to add the value of the pixel above as well
            secondSum = tempSum + sumImage[row-1][col];
            //store the current sum value inside sumImage
            sumImage[row][col] = secondSum;
            secondSum = 0;
```

- Then I wrote a for-loop to go through each row and column and keep summing the pixel values to populate our sumImage. Note that as we had discussed before - when we go to the 2nd row onwards we just need to add the values on the left side and the value above it from sumImage.



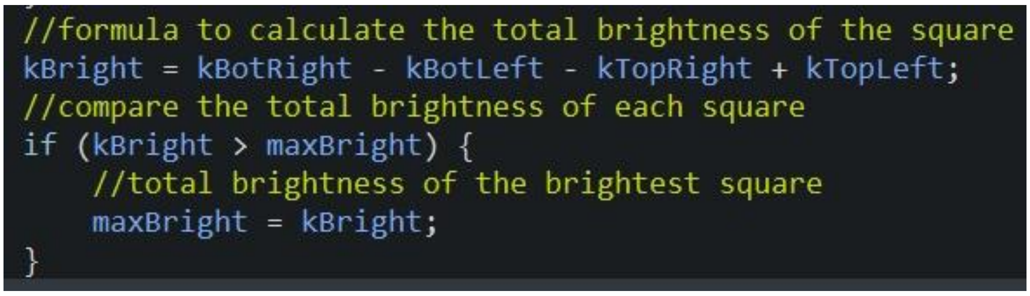
```
//we start from the pixel on the bottom right of Square
for (int rowx = (k-1); rowx < rowsize; rowx++) {
    for (int colx = (k-1); colx < colsize; colx++) {
        //pixel on the bottom right of square
        kBotRight = sumImage[rowx][colx];
        //if we have column on the left of square
        if((colx > k-1)) {
            //pixel outside bottom left corner of square
            kBotLeft = sumImage[rowx][colx-k];
        }
```

- Next once we have created our sumSquare(that is our summed-area table). We can use in to find the value of our k by k square each time we move it.
- As discussed in our examples before when we first start the k by k square is on the top left of our image and we can just take the pixel from the sumImage that corresponds to the bottom right of our k by k square. (I have named this kBotRight).



```
//if we have row above the square
if ((rowx > k-1)) {
    //pixel outside the top right of square
    kTopRight = sumImage[rowx-k][colx];
}
//if we have row above and column on the left of square
if ((rowx > k-1)&&(colx > k-1)) {
    //pixel outside the top left of the square
    kTopLeft = sumImage[rowx-k][colx-k];
}
```

- Similarly, based on the position of the square, I find out kBotRight, kTopRight, kTopLeft and kBotLeft.
- Each time the for loop executes, the square shifts once. At the end of each loop we use the below formula to find out the total brightness of the current square.



```
//formula to calculate the total brightness of the square
kBright = kBotRight - kBotLeft - kTopRight + kTopLeft;
//compare the total brightness of each square
if (kBright > maxBright) {
    //total brightness of the brightest square
    maxBright = kBright;
}
```

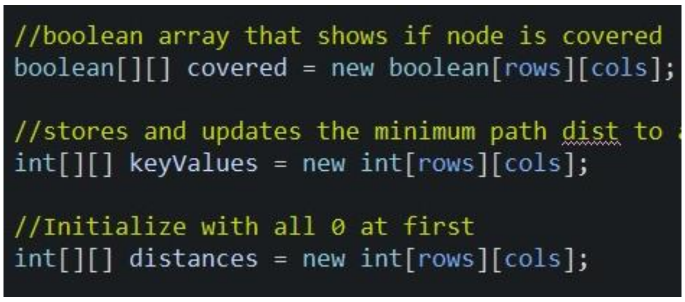
- Then we compare the total brightness of each square with the previous square and keep a record of the brightest total value.
- We return maxBright at the end which is the maximum total value of the brightest k by k square.

<u>Complexity Analysis</u>

- We take a look at each pixel in the image while making sumImage.
- Besides that, if the size of our k by k square is 1 then we need to go through each pixel in the sumImage. (In the worst case).
- (R-K+1)*(C-K+1) is the general complexity. In the worst case if K = 1 then we get O(P).
- Therefore, our time complexity is O(2P) where P is the number of pixels, can also be written as **O(P).**
- In terms of memory required, an image with P pixels requires memory of O(P). sumImage is the same size as our image and hence also requires memory of O(P). Therefore, the total amount of space/memory we will require is O(2P).

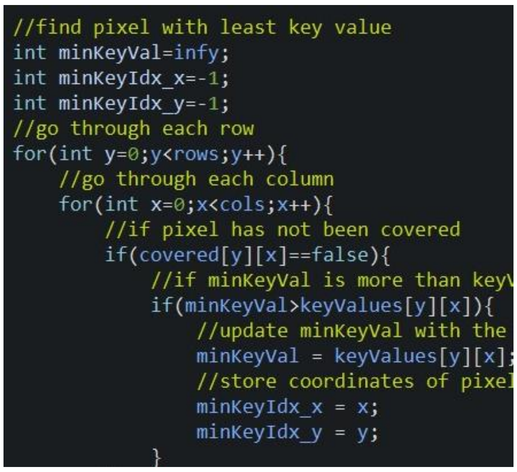
Darkest-Path Explanation

- In the Darkest-Path problem, we need to find the darkest path between two pixels and we will return the maximum pixel brightness in that path.
- After a bit of thinking, I decided to make use of Dijkstra's algorithm, but I had to modify it in such a way that the total distance of any path is the maximum brightness that was encountered while taking that path.
- First, I imagined that each pixel is like a node and so each pixel is connected to the pixels above, below, and beside it. The edge between the two pixels is going to be the maximum brightness between the two pixels.
- Now, since we have decided how our pixels are connected, what the value of the edges are how distance between two points will be calculated – we can proceed to write our program.
  This page on Wikipedia about Dijkstra algorithm was extremely helpful and had some pseudo codes that helped me imagine how to begin writing my program:
  https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- At the start, I am going to create 3 arrays called covered, keyValues and distances. They are the same size as my image.



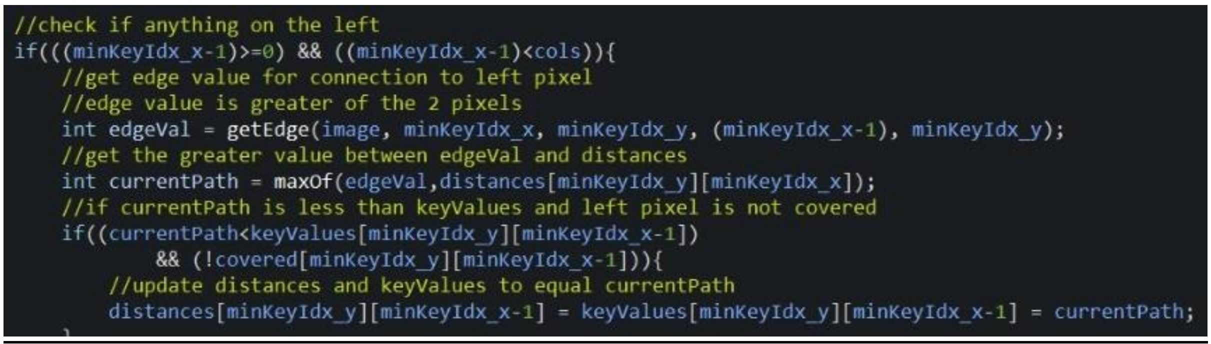
```
//boolean array that shows if node is covered
boolean[][] covered = new boolean[rows][cols];

//stores and updates the minimum path dist to 
int[][] keyValues = new int[rows][cols];

//Initialize with all 0 at first
int[][] distances = new int[rows][cols];
```

- I am going to use covered to record which pixels have already been processed.
- keyValues will store the minimum path distance to each of the pixel from the start pixel. We must remember here that we are not adding the distance here – we are just taking the maximum brightness the path has and setting that as the distance for the whole path.
- We will also have an array called distances. We will see after this how and why we are using this array.
- We must also, populate the 3 arrays we created above – covered will be all False at first, keyValues will be infy (infy is just 1000 in this case. Any value above 255 would work.) Distances is filled with 0.
- We set keyValues corresponding to our start pixel as 0 because the distance from the start pixel to itself is 0.
- The value of the start pixel must be stored inside distances array. We will see why later.
- We then begin our while-loop where we will perform all our main operations. The while-loop will keep executing until the end pixel has been processed and marked as covered.



```
//find pixel with least key value
int minKeyVal=infy;
int minKeyIdx_x=-1;
int minKeyIdx_y=-1;
//go through each row
for(int y=0;y<rows;y++){
    //go through each column
    for(int x=0;x<cols;x++){
        //if pixel has not been covered
        if(covered[y][x]==false){
            //if minKeyVal is more than key
            if(minKeyVal>keyValues[y][x]){
                //update minKeyVal with the
                minKeyVal = keyValues[y][x];
                //store coordinates of pixel
                minKeyIdx_x = x;
                minKeyIdx_y = y;
            }
```

- We then go through all the pixels that have not yet been covered and choose the pixel with the least keyValue. Once we identify the pixel that has the minimum keyValue, we store its coordinates. (the start pixel will be chosen when the program first executes).
- Then we will mark the selected pixel as covered. After that we can begin processing the pixel.
- I have then written 4 if-conditions to check if there are pixels above/below/beside the current pixel and process them accordingly. Since all the 4 if-conditions are very similar, I will discuss one of them in detail.



```
//check if anything on the left
if(((minKeyIdx_x-1)>=0) && ((minKeyIdx_x-1)<cols)){
    //get edge value for connection to left pixel
    //edge value is greater of the 2 pixels
    int edgeVal = getEdge(image, minKeyIdx_x, minKeyIdx_y, (minKeyIdx_x-1), minKeyIdx_y);
    //get the greater value between edgeVal and distances
    int currentPath = maxOf(edgeVal,distances[minKeyIdx_y][minKeyIdx_x]);
    //if currentPath is less than keyValues and left pixel is not covered
    if((currentPath<keyValues[minKeyIdx_y][minKeyIdx_x-1])
            && (!covered[minKeyIdx_y][minKeyIdx_x-1])){
        //update distances and keyValues to equal currentPath
        distances[minKeyIdx_y][minKeyIdx_x-1] = keyValues[minKeyIdx_y][minKeyIdx_x-1] = currentPath;
```

- Above you can see the first if-condition that checks if there is a pixel on the left side. If it detects a pixel on the left it first finds out the value of the edge between two pixels. We had discussed that the edge between two pixels is just the greater value/brightness of the two pixels. (function getEdge just compares 2 pixels and returns the greater value).
- currentPath compares the edge value to the value we had stored inside our distances array and takes the higher value. This is because even if the edge value between two pixels in the image is for example 0 we must also see the maximum brightness we encountered while taking that path and hence set the brightness of that path as the higher value.
- That is why in the next step we check if currentPath is less than the stored keyValue and the left pixel is not yet processed then we update the keyValue and the distances value of the left pixel to currentPath.
- Note that if we see that the left pixel is already processed(marked as covered) or that the currentPath we calculated is more than or equal to the existing keyValue of that pixel – then we do not modify the existing keyValue or distances array value. (If we had already found a way to reach that pixel with a lower brightness then we do not want to modify it).
- And the other 3 if-conditions keep checking the pixels on the other sides and at the end we go back at the start of the while-loop which again repeats the process by choosing the pixel with the least keyValue.
- I have drawn a few pictures on the next page which will help to better explain how my program works.

0	0	85	85	85
0	85	85	170	170
170	255	255	0	170
170	170	170	170	170

- Suppose we have the image on the right and we want to go from 85 to 170. We can observe below how the each time the while-loop executes, we choose the pixel with the minimum keyValue and begin checking the pixels on each side.

covered

FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

keyValues

1000	85	1000	1000	1000
85	0	85	1000	1000
1000	255	1000	1000	1000
1000	1000	1000	1000	1000

distances

0	85	0	0	0
85	85	85	0	0
0	255	0	0	0
0	0	0	0	0

minkeyVal	0
minKeyIdx_x	1
minKeyIdx_y	1

covered

FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

keyValues

85	85	85	1000	1000
85	0	85	1000	1000
1000	255	1000	1000	1000
1000	1000	1000	1000	1000

distances

85	85	85	0	0
85	85	85	0	0
0	255	0	0	0
0	0	0	0	0

minkeyVal	85
minKeyIdx_x	1
minKeyIdx_y	0

covered

TRUE	TRUE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

keyValues

85	85	85	1000	1000
85	0	85	1000	1000
1000	255	1000	1000	1000
1000	1000	1000	1000	1000

distances

85	85	85	0	0
85	85	85	0	0
0	255	0	0	0
0	0	0	0	0

minkeyVal	85
minKeyIdx_x	0
minKeyIdx_y	0

- Notice how we start at the pixel at (1,1) then it updates the 4 pixels next to it. Then the next time the while-loop runs it chooses the minKey value out of the pixels that are not already covered. (it chooses the first 85 it encounters).
- The while-loop keeps running until we manage to process our end-pixel in which case it checks the maximum brightness we encounter on our path to the end pixel (this is stored inside the distances array) and then returns it.

covered

TRUE	TRUE	TRUE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

keyValues

85	85	85	85	1000
85	0	85	1000	1000
1000	255	1000	1000	1000
1000	1000	1000	1000	1000

distances

85	85	85	85	0
85	85	85	0	0
0	255	0	0	0
0	0	0	0	0

minkeyVal	85
minKeyIdx_x	2
minKeyIdx_y	0

covered

TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

keyValues

85	85	85	85	85
85	0	85	170	1000
1000	255	1000	1000	1000
1000	1000	1000	1000	1000

distances

85	85	85	85	85
85	85	85	170	0
0	255	0	0	0
0	0	0	0	0

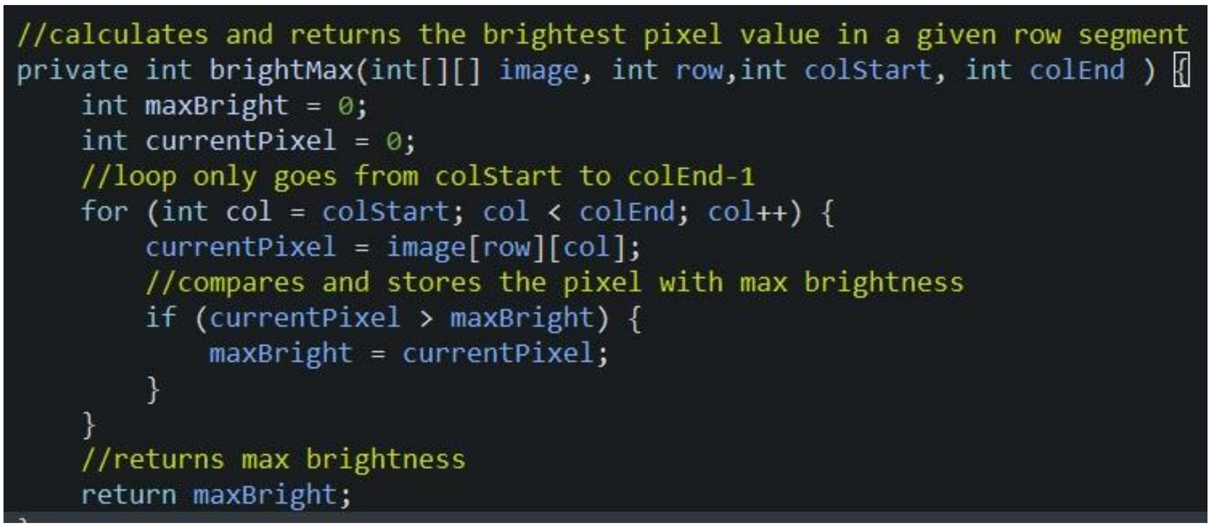
minkeyVal	85
minKeyIdx_x	3
minKeyIdx_y	0

Complexity Analysis

- Our program creates 3 arrays, each of which take O(P) time to make, other than that for each pixel we process, we need to find the minKey value. To find the minKey value we go through the entire keyValues array.
- So, we have O(3P) plus O(P^2) which is the same as **O(P^2).**
- In terms of memory complexity, we created 3 arrays each of size P, that gives is O(3P).

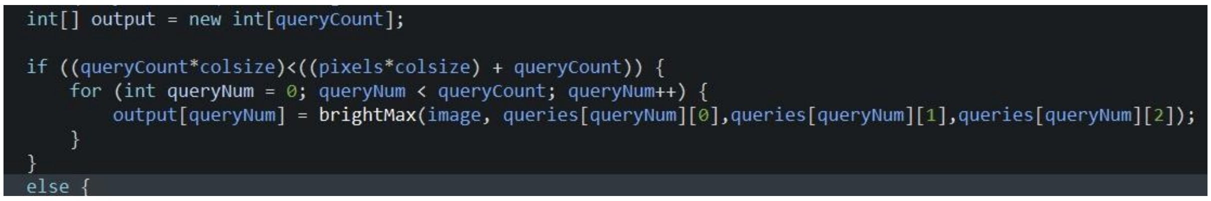
Brightest-Pixels-InRowSegments Explanation

- For the fourth problem, I first wrote a function to output the brightest pixel value based on a given query. Note that I only need to go through the columns in that single row



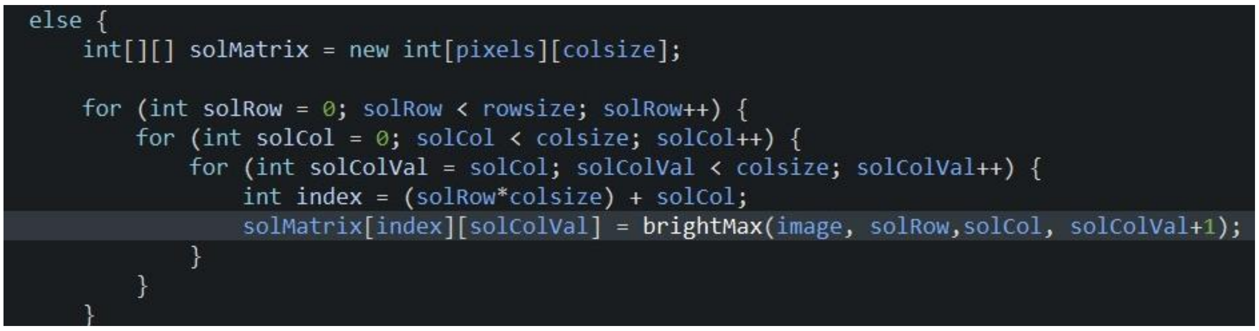
```
//calculates and returns the brightest pixel value in a given row segment
private int brightMax(int[][] image, int row,int colStart, int colEnd ) {
    int maxBright = 0;
    int currentPixel = 0;
    //loop only goes from colStart to colEnd-1
    for (int col = colStart; col < colEnd; col++) {
        currentPixel = image[row][col];
        //compares and stores the pixel with max brightness
        if (currentPixel > maxBright) {
            maxBright = currentPixel;
        }
    }
    //returns max brightness
    return maxBright;
```

- Now I began writing my main function that goes through each query.



```
int[] output = new int[queryCount];

if ((queryCount*colsize)<((pixels*colsize) + queryCount)) {
    for (int queryNum = 0; queryNum < queryCount; queryNum++) {
        output[queryNum] = brightMax(image, queries[queryNum][0],queries[queryNum][1],queries[queryNum][2]);
    }
}
else {
```

- At first, I was simply going through each query, calculating the brightest pixel in that row segment using the function I had written earlier and storing my answer to each query inside the output array.
- The complexity of my program now was O(QC) where Q is the number of queries and C is the column size.
- However, I realised that if the number of queries exceed the number of pixels then QC becomes too high and my program is not so efficient anymore.
- So, to solve this problem I added an if-condition to check if (queryCount*columnsize) exceeds (pixels*columnsize) + queryCount then I go to my else statement when I am now going to explain.



```
else {
    int[][] solMatrix = new int[pixels][colsize];

    for (int solRow = 0; solRow < rowsize; solRow++) {
        for (int solCol = 0; solCol < colsize; solCol++) {
            for (int solColVal = solCol; solColVal < colsize; solColVal++) {
                int index = (solRow*colsize) + solCol;
                solMatrix[index][solColVal] = brightMax(image, solRow,solCol, solColVal+1);
            }
        }
    }
```

- If the program goes into the else-condition, then I create a solution matrix. Basically what I want to do is store every possible solution that a query can have inside a 2D array so after than regardless of how huge my number of queries is I can just lookup my solution-matrix and find the answer to the query. (Instead of each time having to go through each column).
- To better explain how I plan to calculate and store the solution to each possible query, I have drawn a few pictures –

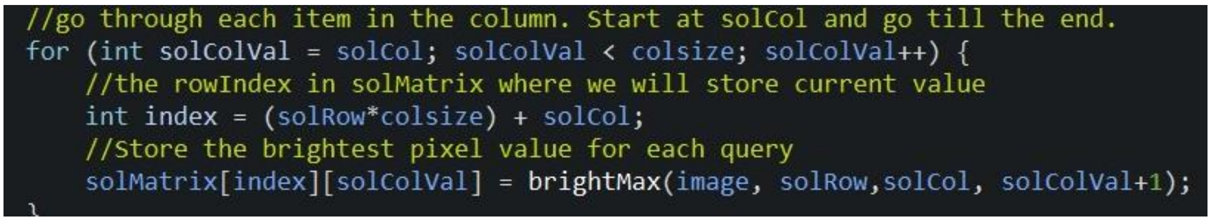
	0	1	2	3	4
0	0	0	85	85	85
1	0	85	85	170	170
2	170	255	255	0	170
3	170	170	170	170	170

	0	1	2	3	4	
0	0	0	85	85	85	
1		0	85	85	85	
2			85	85	85	row1
3				85	85	
4					85	
5	0	85	85	170	170	
6		85	85	170	170	
7			85	170	170	row2
8				170	170	
9					170	
10	170	255	255	255	255	
11		255	255	255	255	
12			255	255	255	row3
13				0	170	
14					170	
15	170	170	170	170	170	
16		170	170	170	170	
17			170	170	170	row4
18				170	170	
19					170	

- Imagine that we have the above 2D image. The image has 4 rows and 5 columns.
- I am going to create an array called solMatrix of size (4 x 5) rows and 5 columns.
- Now I am going to go through each and every value a query can output for every row and store the results in my solutionMatrix.
- So if we look at the first row then if we start at the first 0 and end at the first 0 then the brightest pixel is 0 so we store a 0 inside solMatrix.
- Next, if we start at the first 0 can calculate the brightest pixel output if we end at the 0 in the second column then 85 in the third column and so on then we finally get the first row of solMatrix.
- Next we start at the second 0 and repeat the whole process again. Note that the end must always come after the start so that Is my I left the pixels in yellow blank. So, each time we shift the start and calculate the brightest pixel for each end value – we can keep storing the values in solMatrix.
- After we finish with the first row in the image, we can proceed to the second row and repeat the process. As you can see on the right – storing the possible solutions for each row in the image requires 5 rows in our solMatrix. This number will obviously change depending on the size of the input image we are given.
- Once we have created our solMatrix, finding the answer to a given query is relatively easy we just have to reverse what we did earlier and extract the answer to a given query from solMatrix.

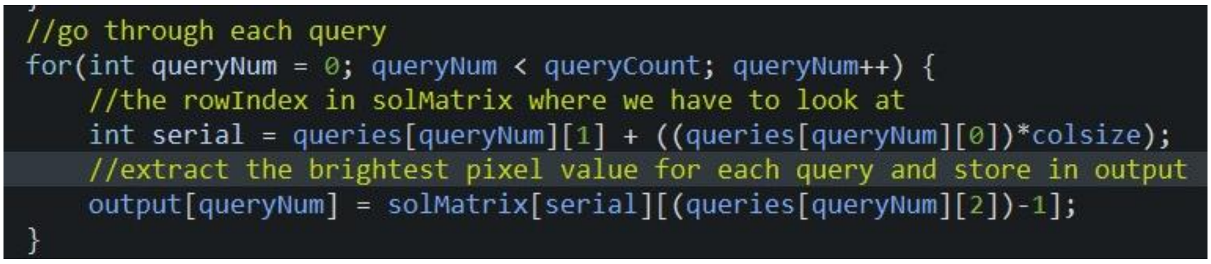
- Notice how the process to get the answer to a query is exactly the opposite of what we had done while storing results inside solMatrix.

While Storing –



```
//go through each item in the column. Start at solCol and go till the end.
for (int solColVal = solCol; solColVal < colsize; solColVal++) {
    //the rowIndex in solMatrix where we will store current value
    int index = (solRow*colsize) + solCol;
    //Store the brightest pixel value for each query
    solMatrix[index][solColVal] = brightMax(image, solRow,solCol, solColVal+1);
```

While Extracting –



```
//go through each query
for(int queryNum = 0; queryNum < queryCount; queryNum++) {
    //the rowIndex in solMatrix where we have to look at
    int serial = queries[queryNum][1] + ((queries[queryNum][0])*colsize);
    //extract the brightest pixel value for each query and store in output
    output[queryNum] = solMatrix[serial][(queries[queryNum][2])-1];
}
```

__Complexity Analysis__

- If (queryCount*colsize)<(pixels*colsize)+queryCount then my complexity will be **O(QC).** Because in the worst case each query will need me to go through the entire column.
- However, If (queryCount*colsize)>(pixels*colsize)+queryCount and I go into my else-condition then I will have to create solMatrix in which I go through each possible value of each possible query.
- The complexity now will be O(PC) for making solMatrix plus O(Q) for the queries. Hence, my total time complexity will be **O(PC + Q).**
- So depending on the condition met I can have 2 possible time complexities –
    - **O(QC).**
    - **O(PC + Q).**
- In terms of memory required, the first condition has very little additional memory requirement. (we will require O(1) memory for storing the brightest pixel in the brightMax function).
- But for the second condition, solMatrix has a size of P * C hence the memory complexity will be O(PC).