CITS2200 Project Report

Flood Fill Count:

The purpose of the floodFillCount function is to return the number of pixels that are to be changed to black when provided a starting pixel and any adjacent pixels (up, down, left, right) that are the same colour (0-255) as the starting pixel.

The implementation I have used is a recursive depth-first search, where the function will recursively traverse through the bordering pixels of the input pixel until either the edge of the image is reached or a pixel of differing colour is reached. The global variable “count” is incremented every time a bordering pixel of the same colour is found and it is noted that this pixel has been counted in a 2D integer array. Therefore every bordering pixel of the starting pixel is searched, as well as any pixels bordering those who border the starting pixel and are the same colour as the starting pixel and so forth. The pixels are also each only counted exactly once. If the starting pixel already has the colour of black then the function will simply return 0. Therefore this method will return an accurate count of the bordering pixels of the same colour.

All recursive traversals which search each node only once have a time complexity of O(n), and in this case n refers to the number of pixels. Therefore the time complexity of this function at worst case is O(Pixels). The function also has to store a 2D array referencing a Boolean value for each pixel as to whether it has been searched or not, this means that it will have a memory complexity of also O(P).

Brightest Square:

The purpose of the brightestSquare function is to return an integer which is the brightness of the brightest k*k square of pixels in the image, given k and the image.

I have implemented a solution to this problem through calculating the brightness of the first k*k square in the top left corner of the image, then looping through and adding the brightness of the next k*1 column, subtracting the brightness of the left-most discarded k*1 column and repeating this for all rows of the image.

Using this solution, the function will therefore calculate the brightness of every k*k square in the image, starting from the top left pixel and finishing at the k*k square in the bottom right of the image. Therefore the function will return an accurate integer for the brightness of the brightest square.

The time complexity of this function is O(Pk). In the worst case, the function needs to search each pixel k times whilst new rows are added, and another k times for whilst the rows are being added. Therefore each pixel will be searched Pixels*2*k times, and with 2 being a constant this simplifies to Pk for large values. The function only stores the integer brightness of the last column of the square and the integer brightness of the brightest square seen so far, and therefore memory complexity is negligible.

Darkest Path:

The purpose of the darkestPath function is to find and return the integer brightness of the darkest path from a given starting pixel to a given end point pixel. The path follows any pixels which border the previous pixel (up, down, left, right).

The solution implemented uses a variation of Dijkstra's path finding algorithm. Initially all paths relative to the starting pixel have a brightness equal to the maximum value of an integer and then the method will search every adjacent pixel to the current pixel, relaxing and storing the minimum brightness path that can be used to reach that pixel so far in a 2D integer array and then using this value to determine whether a darker path can be found to its adjacent pixels using this value. When a path of lower brightness is found, relative to the starting pixel, that pixel is then added into a priority queue so that all of its neighbouring pixels are also searched and so on. Therefore, the method will store the integer minimum brightness of a path from the starting array to every pixel in the image, in a 2D integer array, and then finally return the value of this array at the location of the given end pixel.

The time complexity of Dijkstra's algorithm on a graph with V vertices and E edges is O(V + E*logV). Therefore, with the implementation of Dijkstra's algorithm used, the darkestPath function has a time complexity of O(P*logP), where P = Pixels. It is also worth noting that the function has to store the path brightness integers for each pixel relative to the starting pixel, and therefore needs a 2D array of size P. Therefore darkestPath also has a memory complexity of size O(P).

Brightest Pixel In A Row Segment:

The purpose of the brightestPixelsInRowSegments function is to return a one dimensional integer array of results referring to the integer brightness of the brightest pixel in the given row segments, where the results are stored in the resulting array in the same order as the queries are given.

The implemented solution to this problem simply uses a fixed length loop, with length equal to the number of queries provided. Within each iteration of this loop the method will use a nested fixed length loop to search through each pixel in the specified, fixed length, row segment, storing the highest integer brightness that has been seen so far in the corresponding space in the result integer array. Therefore, with all pixels being searched for each segment this method will always return the value of the brightest pixel for each row segment. Since each pixel in the row segment would have to be searched at least once to form a one dimensional array for a quicksort, time can be saved through simply searching each pixel once and finding the maximum of the searched pixels.

The time complexity of this implementation is equal to the number of pixels in the row segment for each query, therefore in a worst case situation where the row segments are the full width of the rows (or equal to the number of columns, C) the time complexity of the function is O(QC), where Q = Number of queries.