Homework #4

Question #1

```
Function findMalfunctioningFuse (fuses)
    Start = 0
    End = lengthth(fuse) - 1
    while Start <= END                          log n
        MID = (START + END) // 2  ⟶  Decrease and conquere
                                      constant factor 2
        IF fuses[MID] is healthy
            START = MID + 1

        ELSE
            END = MID - 1


    RETURN START

END FUNCTION
```

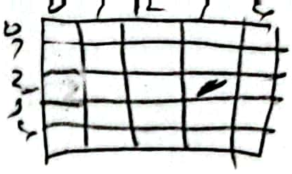
Algorithm Description

1) Start in the middle: Begin by checking the fuse in the middle of the sequence. If this fuse is healthy, it implies that the malfunctioning fuse must be in the second half of the sequence. If it's not healthy, the problem lies in the first half.

2) Halve the problem: Depending on where the problem is suspected to be, halve the sequence of fuses to be checked next.

3) Iterate checking: Repeat the process of checking the middle fuse of the current subset and halving the subset accordingly, until you find the broken fuse.

- Time Complexity Analysis
- The algorithm divides the problem space in half with each iteration
- If n is the number of fuses, the algorithm takes $\log_2(n)$ steps in the worst case to find the malfunctioning fuse.
- Therefore, the time complexity of the algorithm is $O(\log n)$



1

question #2

```
Function findBrightestPixel (grid):
 left = 0
 right = width of grid - 1
 top = 0
 bottom = height of grid - 1

 while left <= right and top <= bottom:
     midcol = (left + right) / 2
     midrow = (top + bottom) / 2
     brightestPixel = getBrightestPixel (midRow, midcol, grid)
     if (isBrightestAmongNeighbors (brightestPixel, grid):
        return brightestpixel

     if brightness of top neighbor > brightestPixel:
        bottom = midRow - 1
     else if brightness of bottom neighbor > brightestPixel:
        top = midRow + 1
     elseif brightness of left neighbor > brightestpixel:
        right = midcol - 1
     else if brightness of right neighbor > brightestPixel:
       left = midcol + 1

  return None


Function getBrightestpixel (row, col, grid):
  brightest = grid [row] [column]

   if row > 0:
      brightest = max (brightest, grid [row - 1] [column])

   if row < len (grid) - 1:
       brightest = max (brightest, grid [row + 1] [column])
   if column > 0
       brightest = max (brightest, grid [row] [column - 1])
   if column < len (grid[0] - 1:
        brightest = max (brightest, grid [row] [column + 1])
  return brightest
```

```
function isBrightestAmongNeighbors (brightestPixel, grid):
    row = pixel.row
    column = pixel.column

    if row > 0 and grid[row-1][column] >= pixel brightness:
      return false
    if row < len(grid) -1 and grid[row+1][column] >= pixel brightness
    if column > 0 and grid[row][column-1] >= pixel brightness:
       return false
    if column < len(grid[0]) -1 and grid[row][column+1] >=
                                                       pixelbrightness
       return false

    return True.
```

Algorithm

1. Initialize: Start with the entire grid of the search space

2. Find midpoint: In each iteration find the midpoint row and column of the current search space

3. check Neighbors: Compare the brigness of the midpoints pixel with its immediate neighbors (top, bottom, left, and right). If the midpoint pixel is brighter than all its neighbors return it as the brightest pixel.

4. Narrow Down Search Space: If the midpoint pixel is not the brightest, determine which neighbor is brighter. The search space can be reduced to the quadrant that includes the brighter neighbor indicates the presence of the brightest pixel in that direction.

5. Repeat. Repeat steps 2 to 4 for the new smaller search space until the brightest pixel is found

6. Terminate: The algorithm. when the brightest pixel is found or when the search cannot be reduced further

Time complexity analysis

72 ①

The time complexity of this algorithm can be analyzed as follows

Each iteration approximately halves the search space (either the number of rows or column is halved).

If the grid has dimensions $n \times m$ the number of iterations required to reduce the search space to a single element is $\underline{O(\log n + \log m)}$

Thus the overall time complexity is $\underline{O(\log n + \log m)}$

Question #3

```
Function findMaxTotalArea(f,n):
    maxArea=0
    currentArea=0
    start =0
    end =0

    for  i=0  to  n
         currentArea  +=f(i)

         if currentArea <0:
              currentArea  =0
              start =i+1

        if currentArea > maxArea:
              maxArea=currentArea
              end =i

    return start, end
```

Algorithm Explanation

1. Initialization: The variable 'maxArea' and 'currentArea' are initialized to 0. 'start' and 'end' are used to keep track of the the intervals boundaries.

2. Iterate through the Function value: The algorithm iterates from 0 to n adding the value of f(i) to 'currentArea' at each step. this process accumulates the total area under the function f(x) for current subinterval

3. Reset on Negative Area: If 'currentArea' becomes negative its reset to 0 and the 'start' index is moved to i+1. This step essentially discards the current subinterval, as it decrease the overall area

4. Update maximum Area: If 'currentArea' is greater than 'maxArea' It means the current subinterval has a larger total area than any previously considered intervals. Thus, 'maxArea' is updated to this larger value, and the 'end' index is set to i.

5. Return values: The function return 'start' and 'end' which define boundaries of the subinterval that yields the maximum total area under f(x).

Time Complexity Analysis.

- The function consist of a single loop that iterates over the interval [0, n]. Each iteration involves a constant amount of work: evaluating f(i), updating

- The time complexity of evaluating f(i) depends on the function itself. If we assume that f(i) can be computed in constant time then the complexity of each iteration is $\underline{O(1)}$

- Therefore the overall time complexity of the 'findMaxTotalArea' function is $\underline{O(n)}$ where n is the length of the interval [0,n]

Question #4

```
Function FindMinimumLatencyPath(graph):
    Initialize minLatency to infinity
    Initialize minPath to an empty list

    For each pair of nodes (source, destination) in graph:
        Initialize path to an empty list
        Call DFS(graph, source, destination, 0, path, minPath, minLatency)
    return minPath, minLatency


Function DFS(graph, current, destination, currentLatency, path, minPath,
                                                            minLatency):

    Add current node to path
    If current node is destination:
        If currentLatency < minLatency:
            update minLatency
            copy path to minPath

    Else:
        For each neigbor of current node:
            If neigbor is not in path:
                newLatency = currentLatency + latency(current, neighbor)
                DFS(graph, neighbor, destination, newLatency, path, minPath
                                                            , minLatency)
    Remove current node from path
```

1- Find minimum Latency Path (FindMinimumLatencyPath): This function finds the lowest latency path for all pairs of nodes in the network graph. The DFS algorithm is called for each pair.

2- DFS: This function explore all paths starting from the current node until it reaches the destination. At each step, the total latency of the current path is calculated. If the destination is reached and this path's latency is less than the lowest found so far, this path is recorded.
3- Result: The algorithm returns the path the lowest latency and its latency for all node pair

Time Complexity

Considering N as the number of nodes and E as number of edges in a graph we can say that DFS has a time complexity of $O(N)$ for each node as in the worst case it may need to visit all nodes. However, Since this algorithm runs DFS for every pair of nodes the total time complexity becomes $O(N^2 \times N) = O(N^3)$ This represent a significant computational load depending on the density and the number of nodes in graph.

$O(N^3)$

question #5

```
Function findResourcesExtremes(tasks):
    IF tasks has only one Element:
        Return (tasks[0], tasks[0]) as both max and min
    Else
        mid = length of task / 2                                          $T(\frac{n}{2})$
        (leftmax, leftmin) = findResourceExtremes(tasks[0:mid])           $T(\frac{n}{2})$
        (rightMax, rightmin) = findResourceExtremes(tasks[mid:end])
        maxTask = max(leftMax, rightMax, key=lambda task: task.resources)
        minTask = min(leftMin, rightMin, key=lambda task: task.resources
        return (maxTask, minTask)
```

Algorithm Explanation

- The algorithm work by recursively splitting the task list and finding the max and min in each half
- These max and min tasks are then compared against each other to find the overall max and min.
- The base case is when the task list has only one task in which case that task is both the max and the min.
- The key idea is to reduce the number of comparisons made, making algorithm efficient for large lists of tasks.

Time complexity Analysis

1) Base Case: When $n = 1$ the function simply returns the task as both the maximum and minimum so $T(1) = O(1)$

2) Divide Step: The list of tasks is divided into two halves each of size approximately $\frac{n}{2}$

3: Recursive Calls: Two recursive calls are made, one for each half. Therefore, the work done by these calls can be represented as $2T(\frac{n}{2})$

4- Combine step: After getting the results from the recursive calls (leftMax, leftMin, rightmax, rightMin) the algorithm finds the overall maximum and minimum. This step involves constant time comparissions so it is $O(1)$

$T(n) = 2T(n/2) + O(1)$ use master Theorem

$a = 2$

$b = 2$

$f(n) = O(1)$

$$T(n) \quad \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon} \end{cases}$$

we use 1. case

$$T(n) = \Theta(n^{\log_2 2}) = \underline{\underline{\Theta(n) \quad = O(n)}}$$

CamScanner ile tarandı