20210420066
Ahmet Yigit

Q1-)

```
def findFlawedFuse(fuses, NORMAL_CURRENT = 4):

    length = len(fuses)
    i = length-1

    while i >= 0:
        if fuses[i] == NORMAL_CURRENT:
            if i == length-1:
                return None
            return i+2
        i -= 1
    return -1
```

Time Complexity:

- The loop starts at the last element 'i=length-1' and iterates backward until it reaches the first element 'i=0'. So it goes as length of fuses. It takes O(n) time.
- Inside loop, there is only one comparision so it takes O(1) times.
- Return sides get only O(1) time to.

=> Therefore, the overall time complexity of this function is O(n) where n is the length of fuses.

$$\underline{\text{Time Comp} = O(n)}$$

Q2-) let's say

$n = \text{len}(grid)$ $\quad$ $m = \text{len}(grid[0])$

Time complexity analysis

↳ Iterating through grid

↳ The nested loops iterate through the grid, excluding the borders ('1' to len(grid)-2' and '1' to len(grid[0])-1, so this means that $(n-2)*(m-2)$ it is approximately same thing $n*m$

↳ Also it has constant time operations like conditions.

↳ Iterating loops in border

↳ Each grid there is 4 loop for 2 border row and 2 border colum and each one takes $O(n)$ and $O(m)$

↳ Controlling corner

↳ It includes only constant comparisons. $O(1)$

↳ So time complexity depends on 'Iterating through Grid' which is approximately $O(n*m)$.

Time complexity = $O(n*m)$

8B)

```
def findMaxTotalArea(areaSize):

    result = -9999999 -1
    tempValueOfMax = 0

    for i in range(0, areaSize):
        tempValueOfMax += f(i)

        result = max(result, tempValueOfMax)

        if tempValueOfMax < 0:
            tempValueOfMax = 0

    return result
```

Time Complexity Analysis:

→ Invoke f function and it takes constant time. O(1)
  ↳ it takes x and returns depth/value of xth element.

→ findMaxTotalArea

→ The loop iterates areaSize times, from 0 to areaSize. So it takes O(n) time.
  n = areaSize

→ Inside loop

  ↳ calling f function and add to variable takes O(1) constant

  ↳ getting max number between 2 number is one comparison and takes O(1) time.

  ↳ comparsion tempValueOfMax takes constant time O(1).

→ returning value is O(1) constant time.

→ So, the loop iterates n tims and dominates the time complexity, and all comparsions are constant time in loop. Overall time complexity is O(n).

time complexity = O(n)

@4-) def exhaustive_search(graph, source, dest):
    # DFS search
    def DFSUtil(currNode, currWayLen, currPath, visited, minWayGraph, minWay=graph):

        if currNode == dest: # base case
            if currWayLen < minWay:
                minWay = currWayLen
                minWayGraph[:] = currPath[:]
            return

        visited.add(currNode)

        for neighbour, exp in graph.graph[currNode]:
            if neighbour not in visited:
                DFSUtil(neighbour, currWayLen + exp, currPath+[neighbour], visited, minWayGraph)

        visited.remove(currNode)

    def dfs(currNode, currWayLen, currPath):
        minWayGraph = []
        visited = set()
        DFSUtil(currNode, currWayLen, currPath, visited, minWayGraph)
        return minWayGraph

    minWayGraph = dfs(source, 0, [source])
    return minWayGraph

- Solution / Approach

↳ Exhaustive search, also known as brute-force search or exhaustive enumeration that is systematically searching all possible solutions to a problem to find the best one.

- Time Complexity Analysis

↳ Exhaustive_Search calls dfs function and return value which is O(1).

↳ dfs time complexity

↳ for each node, the algorithm explores its adjacent nodes.
↳ Let's say that there is 'V' vertices and 'E' edges in the graph.
↳ In worst case, where every node is connected to every other node, the time complexity is DFS will be O(V+E)

⇒ So overall complexity will be $\underline{O(V+E)}$.

05-)

```
def getMinMaxUsingDivideAndConquer(tasks):   T

    if not tasks:
        return None
    if len(tasks) == 1:
        return tasks[0], tasks[0]

    mid = len(tasks) // 2

    lMin, lMax = getMinMaxUsingDivideAndConquer(tasks[:mid])      T(n/2)

    rMin, rMax = getMinMaxUsingDivideAndConquer(tasks[mid:])      T(n/2)

    minTask = min(lMin, rMin)
    maxTask = max(lMax, rMax)

    return minTask, maxTask
```

O(n)

O(n)

→ Time Complexity Analysis

↳ In each function call it call itself by reducing tasks divide by 2.

↳ It can be solve using recurrence relation.

$$T(n) = 2\,T(n/2) + O(n)$$

$$= 2\left[2\,T(n/2^2) + \frac{n}{2}\right) + n$$

$$T(n) = 2^2 + (n/2^2) + n + n$$

$$= 2^2\left\{2\,T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right\} + 2n$$

$$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + 3n$$

$$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$$

$T\left(\frac{n}{2^2}\right) = T(1)$

$\frac{n}{2^2} = 1 \rightarrow n = 2^k$

k^{th} $\quad$ so, $n = 2^k$

$k = \log n$

$$T(n) = 2^2\,T(1) + 2n$$

$$= n + n\log n$$

$$O(n\log n)$$

Time Complexity is $\Rightarrow O(n\log n)$