Emre Oztun
200104004099

1. → The algorithm checks for fuses in indexes 10, 20, 30, ... so that 10 elements are validated to work properly each time

→ If there is no electricity in one of them it searches through the 10 elements to find the broken one.

* Problem size decreases-by-10 after each iteration.

Worst-case: The broken fuse is the last element.
Let assume $n = 10k$ $(k \geq 0)$:

→ There are $n/10$ checks and 10 additional checks to find the broken one after last iteration.

So, $W(n) = \frac{n}{10} + 10 \in \Theta(n)$

↳ i-) $W(n)$ is non-decreasing
ii-) n is " "
iii-) $\Theta(n)$ is 2-invariant.

$W(n) \in \Theta(n)$ for all n.
Therefore, $W(n) \in O(n)$ //

2. Firstly, the algorithm iterates through the inner elements with indexes in range [1, rowNum-1][1, colNum-1].
It checks if these elements are greater than their four immediate neighbours.

↳ There are $(n-2).(m-2)$ iterations and constant operation for checks in each. — $O(m.n)$

→ Secondly, it checks the most outer rectangle without corners by checking if they are greater than three immediate neighbours.

↳ There are $2(n-2) + 2(m-2)$ iterations. — $O(m+n)$

→ If the unique pixel is still not found, then it checks corners.

↳ $O(1)$

* Problem-site decreases-by-one after each pixel check.

$T(n) = O(m.n) + O(m+n) + O(1) \in O(m.n)$ //

(3.) The algorithm iterates through the whole array Starting from lower bound to upper bound.

-> It keeps maxSum and curSum.

curSum: The elements are summed until the curSum is negative since there is no need to include the next elements into the current subsequence of elements after this point.
So, after the point curSum is negative the curSum is resetted to 0 and new subsequence started with the next element.

maxSum: It keeps the maxSum and it's updated by comparing with the curSum.

-> Also; it keeps the indexes curSumLeft/Right index, maxSumLeft/Right index.

Therefore, there are $\underline{n}$ iterations and constant operations in each.
So, $w(n) \in \Theta(n)$ //

(4.) -> The algorithm firstly initialized visited array. — $\Theta(V)$

-> Then in recursive helper method it tries all possible paths starting from source node.

-> If the target node is reached, it compares it with global minLatency variable and copies the path if necessary.

-> In worst case, each vertex is connected to each other. And there is a copy in the last call every time.

-> The start vertex is always source node, and the last vertex is always target node. So, we should consider selecting how many vertex is in a path, and the visit order could chage when we select some set of vertexes. There are (V-2) nodes remaining except source-start.

$$T(n) = \left[\binom{V-2}{1} \cdot 1! \cdot (\overset{\text{visited vertex number}}{1} + \overset{\text{copy path}}{1})\right] + \left[\binom{V-2}{2} \cdot 2! \cdot (2+2)\right] + \dots$$

$$\left[\binom{V-2}{V-2} \cdot (V-2)! \cdot (V+V)\right]$$

$* \binom{a}{b} \cdot b! = a.(a-1) --- (a-b+1)$

$T(n) = (V-2).2 + (V-2)(V-1).4 + -- + (V-2)!.2V$

Each term $< (V-2)!.2V \Rightarrow T(n) < (V-2)!.2V.(V-2)$

Therefore, $T(n) \in O(V^2.V!)$ //

5. → The algorithm divides the array into two in each time end combines the results.

→ Let $n = 2^k$ format:

$T(n) = 2T(n/2) + 1$

By Master's Theorem: $\log_b a = \log_2 2 = 1 \rightarrow n^1$

$f(n) = n^0$

Therefore, $T(n) \in \Theta(n)$

↳ i-) $T(n)$ is non-decreasing ✓

ii-) n " " " ✓

iii) $\Theta(n)$ is θ-invariant ✓

So, $T(n) \in \Theta(n)$ holds for all n.

And $T(n) \in O(n)$ //