CSE 321 HW4
Elifnur Kabalcı
1801042617

1) Algorithm find_flawed_fuse (fuses)

```
    # Input: 'working' or 'broken'
    # Output: index of the broken fuse

    for i in range (0, len(fuses)):
        if fuses[i] == 'broken':
            return i
    return -1      # if no broken fuse is found
```

⇒ We start from the first fuse (index 0) and check each fuse one by one. If we find a fuse that is 'broken', we return its index. If no broken fuse is found after checking all fuses, we return -1.

⇒ The time complexity of this algorithm is O(n), where n is the # of fuses. This is because in the worst-case scenario, we have to check all the fuses, which requires n operation. Therefore, the time complexity is linear with respect to the size of the input. This algorithm is the decrease and conquer approach as it reduces the problem size by a constant (1) at each step by checking one fuse at a time.

2) Algorithm find_Brightest_Pixel (image)

```
    # Input: a 2D grid of pixels
    # Output: coordinates of the brightest pixel
    for i in range (1, len(image) - 1):
        for j in range (1, len (image[i]) - 1):
            if ( image[i][j] > image [i-1][j] and        # top neighbor
                 image[i][j] > image [i+1][j] and        # bottom neighbor
                 image[i][j] > image[i][j-1] and         # left neighbor
                 image [i][j] > image [i][j+1] ):        # right neighbor
                return (i, j)
    return (-1, -1)     # default, cannot find
```

⇒ We start from the top left corner of the image (index (1,1)) and check each pixel one by one. If we find a pixel that is brighter than all of its four immediate neighbours, we return its coordinates. If no such pixel is found after checking all pixels, we return (-1, -1).

⇒ The time complexity of this algorithm is O(nm), where n is the # of rows and m is the # of columns in the image. This is because in the worst case scenario, we have to check all the pixels, which requires nm operations. Therefore, the time complexity is linear with respect to the size of the input. This algorithm follows the decrease and conquer approach as it reduces the problem size by a constant (1) at each step by checking one pixel at a time.

3) Algorithm Find_Max_Area (image)

```
# Input: f(x) over the interval [0,n]
# Output: interval for maximal total area
max_area = 0
start = 0
end = 0
for i in range(0, len(f) - 1):
    area = 0
    for j in range(i+1, len(f)):
        area += f[j]
        if area > max_area:
            max_area = area
            start = i
            end = j
return (start, end)
```

⇒ We start from the first point (index 0) and check each interval one by one. If we find an interval that produces a larger total area than the current maximal total area, we update the maximal total area and the corresponding interval. If no such interval is found after checking all intervals, we return the interval that produces the maximal total area.

⇒ The time complexity of this algorithm is $O(n^2)$, where n is the # of points in the interval [0,n]. This is because in the worst case scenario, we have to check all possible intervals, which requires $n*(n-1)/2$ operations.

4) Algorithm Min_latency_path (graph, source, destination)

```
min_latency = infinity
min_path = []
visited = set()

function DFS(node, path, latency):
    if node == destination:
        if latency < min_latency:
            min_latency = latency
            min_path = path
        return
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            DFS(neighbor, path + [neighbor], latency + graph[node][neighbor])
    visited.remove(node)
DFS(source, [source], 0)
return min_path
```

⇒ We start from the source node and use DFS to explore all possible paths to the destination node. For each path, we calculate the total latency and update the min latency and the corresponding path if the total latency of the current path is less than the min latency. After exploring all paths, we return the path with the min latency.

⇒ The time complexity of this algorithm is $O(n!)$, where n is the # of nodes in the graph. This is because in the worst case scenerio, we have to explore all possible paths, which requires n! operations.

5) Algorithm max_min_resources (tasks)

```
if len (tasks) == 1:
    return tasks[0], tasks[0]
mid = len (tasks) // 2
left_max, left_min = max_min_resources (tasks[:mid])
right_max, right_min = max_min_resources (tasks[mid:])

max_task = left_max if left_max[1] > right_max[1] else right_max
min_task = left_min if left_min[1] < right_min [1] else right_min
return max_task, min_task
```

=) We start with the list of all tasks and divide it into two halves. We recursively apply the same process to each half until we reach the base case where the list contains only one task. Then, we compare the resource demands of the tasks in each half to find the tasks demanding the max and min resources.

=) The time complexity of this algorithm is O(n logn), where n is the # of tasks. This is because in each recursive call, we divide the list of tasks into two halves, which results in a logarithmic number of levels, and at each level, we perform a linear # of operations to compare the resource demands of the tasks