CSE 321 HOMEWORK 4

①
```
def find_malfunctioning_fuse(arr, index):
    if index == len(arr):
        return False
    if arr[index] == 0:
        return index+1
    else:
        return find_malfunctioning_fuse(arr, index+1)
```

Time Complexity Analysis

The function uses a simple recursive approach to traverse the array until it finds a malfunctioning fuse or reaches the end of the array. So it's time complexity is $O(n)$

②
```
def find_brightest_pixel(x, y, arr):
    if check_brightest(x, y, arr):
        return x, y
    if x >= len(arr[0]):
        if y >= len(arr):
            return False
        else:
            return find_bright_pixel(0, y+1, arr)
    else:
        return find_bright_pixel(x+1, 0, arr)
```

Time Complexity Analysis

The recursive function iterates through the 2D array row by row, and for each row, it iterates through the columns. The check for the brightest pixel involves constant time operations, and it is called for each element in the array. So the time complexity is $O(m*n)$ where m is the num of rows and n is the num of columns in the 2D array.

③
```
def sum_of_intervals(arr, first, second):
    if first == second:
        return arr[first]
    else:
        return arr[first] + sum_of_intervals(arr, first+1, second)

def find_largest_area(value_arr, largest, first, second):
    if second == len(value_arr) and first == len(value_arr)-1:
        return largest
    temp = sum_of_intervals(value_arr, first, second)
    if second == len(value_arr)-1:
        first += 1
        second = first
    if temp > largest:
        largest = temp
    return find_largest_area(value_arr, largest, first, second+1)
```

Time Complexity Analysis

"sum_of_intervals" calculates the sum of elements in the given interval. It will iterate through all elements in the array once. So this function's time complexity is $O(n)$

"find_largest_area" iterates through all possible intervals (first and second) in the array. The number of possible intervals is $O(n^2)$

Therefore, the time complexity of this function is $O(n^3)$ because, each recursive call, it performs $O(n)$.

④
```
def exhaustive_search(graph, source, destination):
    minLatency = INFINITY
    minLatencyPath = []
    def DFS(currentNode, currentPathLatency):
        nonlocal minLatency, minLatencyPath
        if currentNode is destination:
            if currentPathLatency < minLatency:
                minLatency = currentPathLatency
                minLatencyPath = currentPath
        for each neighbour of currentNode:
            if neighbour is not in currentPath:
                newLatency = currentPathLatency + latency(currentNode + neighbour)
                DFS(neighbour, newLatency)
    DFS(source, 0)
    return minLatencyPath.
```

Time Complexity Analysis

In the worst case, the algorithm explores all possible paths from the source to the destination.

The branching factor "b" represents the average number of neighbours for each node.

The depth of recursion "d" represents the length of the path from the source to the destination.

Therefore, the time complexity is $O(b^d)$

(5)

```
def DEC (tasks, start, end)
    if start = end:
        return max_task and min_task
    mid = (start + end) // 2
    left_result = DFC (tasks, start, mid)
    right_result = DFC (tasks, mid+1, end)
    max_task = max(left_result [max_task], right_result [max_task], key = get_resource - demand)
    min_task = min (left_result [min_task], right_result [min_task], key = get_resource - demand)
    return max_task and min_task
```

Time Complexity Analysis

At each level of recursion, the algorithm divides the tasks into two halves, and it performs constant work to combine the results from the subproblems. The depth of the recursion is $\log n$, resulting in a total time complexity of $\boxed{O(\log n)}$ (recurrence relation $2T(n/2) + O(1)$)