CSE 321 - Homework 4

1)

```
def find_flawed_fuse(fuses):                    let's say len(fuses) == n
    begin = 0
    while begin < len(fuses):                   } O(n)
        if is_healty_fuse(fuses[begin]):        O(1)
            begin += 1
        else:                                   O(1)
            return begin
    return None

def is_healty_fuse:                             } O(1)
    if fuse == "ok":
        return True
    else:
        return False
```

T(n) = O(n) → average and worst case

T(n) = O(1) → best case

2)

```
def find_distinct_brightest(grid):              → m x n
    row = 0                                     # starting from left corner
    column = 0                                  } O(1)
    row_size = len(grid)
    colum_size = len(grid[0])

    right_brighter = (grid[row][column] < grid[row][column+1])      } O(1)
    down_brighter = (grid[row][column] < grid[row+1][column])
    is_edge = False

    while True:                                 } O(mxn)
        if (not right_brighter and not down_brighter):
            return row, column
        if is_edge:                             } O(1)
            row += 1
            column = 0
        else:
            column += 1
        is_end_of_grid = (row == row_size - 1 and column == column_size-1)
        if is_end_of_grid:
            return None
        is_edge = column == column_size - 1
        if not (column == column_size - 1):
            right_brighter = (grid[row][column] < grid[row][column+1])
        else:
            right_brighter = False
        if not (row == row_size - 1):           } O(1)
            down_brighter = (grid[row][column] < grid[row+1][column])
        else:
            down_brighter = False
```

Worst, average = O(mxn)

best = O(1)

continue

Q3)

```
def find_consecutive_subsets(input_list):          # n
    subsets = []
    n = len(input_list)

    for i in range(n):
        for j in range(i, n):
            subsets.append(input_list[i:j+1])

    return subsets
```

$T(n) = \frac{n(n+1)}{2} \in O(n^2)$

$O(n^2)$

```
def total_area_under_curve(f_values):              # n
    return sum(f_values)
```

$O(n)$

```
def max_area(f_values):
    subsets = find_consecutive_subsets(f_values)   # O(n^2)
    max_area = float("-inf")

    for subset in subsets:                         # n
        area = total_area_under_curve(subset)      # O(n^2)
        if area > max_area:                        # O(1)
            max_area = area

    return max_area
```

$O(n^3)$

$T(n) \cong n^2 + n^3$

$T(n) \in O(n^3)$

worst and average: n — m

Q4)

```
def find_paths(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return [path]
    if start not in graph:
        return []
    paths = []
    for node in graph[start]:
        if node not in path:
            newpaths = find_paths(graph, node, end, path)
            for newpath in newpaths:
                paths.append(newpath)
    return paths
```

DFS

```
def calculate_latency(path, latencies):            # n
    total_latency = 0
    for i in range(len(path)-1):                   # O(n)
        total_latency += latencies[(path[i], path[i+1])]   # O(1)
    return total_latency
```

```
def find_min_latency_path(graph, latencies, start, end):   # V vertices
    paths = find_paths(graph, start, end)          # V! → worst case
    min_latency = float("inf")                     # O(1)
    min_latency_path = None                        # O(1)
    for path in paths:                             # m
        latency = calculate_latency(path, latencies)   # n
        if (latency < min_latency):
            min_latency = latency
            min_latency_path = path
    return min_latency_path, min_latency
```

continue

$O(m \times n)$

$T(n) = m \times n + V!$

$T(n) \in O(V!)$

5)

```
def allocate_resources(tasks):
    if len(tasks) <= 1:                                   O(1)
        return tasks[0], tasks[0]
    mid = len(tasks) // 2
    left_max, left_min = allocate_resources(tasks[:mid])  O(n)
    right_max, right_min = allocate_resources(tasks[mid:])
    return compare_max(left_max, right_max), compare_min(left_min, right_min)   O(1)

def compare_max(task1, task2):
    if task1[1] > task2[1]:                               O(1)
        return task1
    return task2

def compare_min(task1, task2):
    if task1[1] < task2[1]:                               O(1)
        return task1
    return task2
```

$T(n) = 2 \cdot T(n/2) + O(n)$ Solve it by master's theorem.

$a=2 \quad b=2 \quad f(n) = O(n)$

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

$f(n) = O(n^{1})$ so that

$\boxed{T(n) = \Theta(n \log n)}$