February 3, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Algorithm Engineering Lab 00

Azmayen Fayek Sabil, 190042122

Contents

1 Task 1

1.1 Given functions and Big O notation

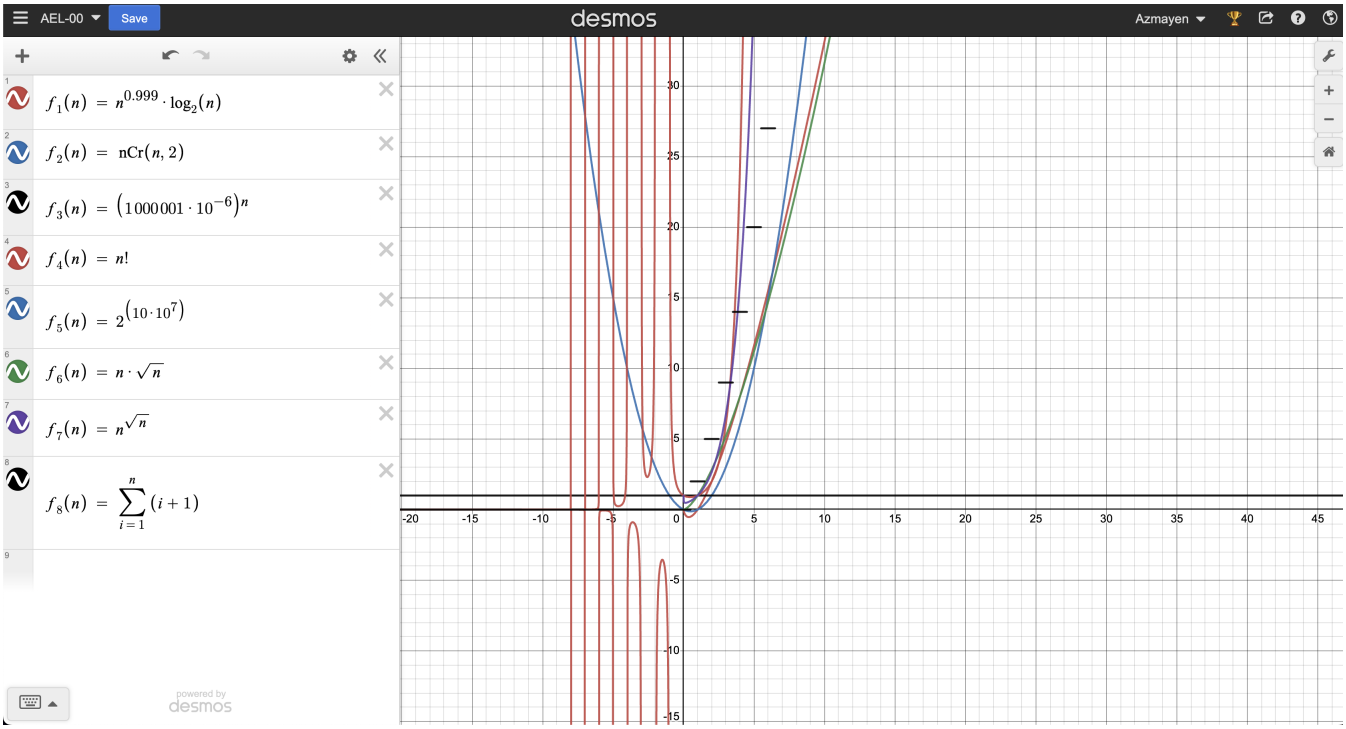
1. $f_1(n) = n^{0.9999} \cdot \log_2(n)$
2. $f_2(n) = \binom{n}{2}$
3. $f_3(n) = (1000001 \cdot 10^{-6})^n$
4. $f_4(n) = n!$
5. $f_5(n) = 2^{10 \cdot 10^7}$
6. $f_6(n) = n \cdot \sqrt{n}$
7. $f_7(n) = n^{\sqrt{n}}$
8. $f_8(n) = \sum_{i=1}^{n}(i+1)$

1.2 Sorted based on Big O notation

The functions in increasing order of asymptotic (big-O) complexity:

1. $f_5(n) = 2^{10 \cdot 10^7} \quad (O(1))$
   Explanation: The function involves an exponential term with a large constant exponent, resulting in exponential growth and a big-O notation of $O(1)$.
2. $f_2(n) = \binom{n}{2} \quad (O(n^2))$
   Explanation: The function represents the binomial coefficient, and its big-O notation is quadratic, as it involves iterating over pairs of elements.
3. $f_1(n) = n^{0.9999} \cdot \log_2(n) \quad (O(n^{0.9999} \log n))$
   Explanation: The function has a logarithmic term and a slightly sublinear exponent, resulting in a big-O notation of $O(n^{0.9999} \log n)$.
4. $f_6(n) = n \cdot \sqrt{n} \quad (O(n^{1.5}))$
   Explanation: The function involves multiplying n by its square root. The big-O notation is $O(n^{1.5})$ since it is a polynomial with a degree between linear and quadratic.
5. $f_8(n) = \sum_{i=1}^{n}(i+1) \quad (O(n^2))$
   Explanation: The function involves a sum of consecutive integers. The sum of consecutive integers up to n is proportional to n^2, leading to a quadratic big-O notation.
6. $f_3(n) = (1000001 \cdot 10^{-6})^n \quad (O((C)^n))$
   Explanation: The function is an exponential with a constant base less than 1, resulting in a big-O notation of $O((C)^n)$.
7. $f_7(n) = n^{\sqrt{n}} \quad (O(n^{\sqrt{n}}))$
   Explanation: The function has an exponent involving $\sqrt{n}$, resulting in superpolynomial growth, specifically $O(n^{\sqrt{n}})$.
8. $f_4(n) = n! \quad (O(n!))$
   Explanation: The function represents the factorial, which grows very quickly. The big-O notation is factorial, denoted as $O(n!)$.

1.3 Graph from Desmos



2 Task 2

2.1 Task 2.1

The given function is:

```
    def find3RDelement(a):
        flag = 2
        for i in range(len(a)):
            if i == flag:
                return a[i]
        print(find3RDelement([1, 2, 7, 4, 5]))
```

Listing 1: Peak Finder 2D

Now, let's analyze the asymptotic (big-O) complexity of the `find3RDelement` function:

Asymptotic Complexity: $O(n)$

Explanation: The function iterates through each element in the input list a using a `for` loop. The loop runs n times, where n is the length of the list. In the worst case, the element to be returned is at the end of the list, making the time complexity linear, $O(n)$.

2.2 Task 2.2

Consider the following Python code:

```
    import numpy as np

    def doingSomethingImportant(p2, sr, sc, prev, new):
        row = len(p2)
        col = len(p2[0]) if len(p2) > 0 else 0

        if sr < 0 or sr >= row or sc < 0 or sc >= col:
            return

        if p2[sr][sc] != prev:
            return

        p2[sr][sc] = new
        doingSomethingImportant(p2, sr - 1, sc, prev, new)
        doingSomethingImportant(p2, sr + 1, sc, prev, new)
        doingSomethingImportant(p2, sr, sc + 1, prev, new)
        doingSomethingImportant(p2, sr, sc - 1, prev, new)

    a2 = [
        [1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 0, 0],
        [1, 0, 0, 1, 1, 0, 1, 1],
        [1, 2, 2, 2, 2, 0, 1, 0],
        [1, 1, 1, 2, 2, 0, 1, 0],
        [1, 1, 1, 2, 2, 2, 2, 0],
        [1, 1, 1, 1, 1, 2, 1, 1],
        [1, 1, 1, 1, 1, 2, 2, 1]
    ]

    x, y = 4, 4
    prev = a2[x][y]
    new = 3
    doingSomethingImportant(a2, x, y, prev, new)

```

```
    a2 = np.array(a2)
    print(a2)
```

Listing 2: Peak Finder 2D

The `doingSomethingImportant` algorithm selectively updates a 2D array by traversing in four directions from a specified coordinate. Two recursive calls maintain a fixed column, moving upward and downward, while the other two maintain a fixed row, moving rightward and leftward.

Asymptotic Complexity: $O(n \times m)$, where n and m are the number of rows and columns, respectively.

Explanation: The algorithm explores the connected component, updating values that meet the specified condition. The worst-case time complexity is proportional to the size of the connected component.

Conclusion: The algorithm efficiently traverses and updates the connected component, with time complexity $O(n \times m)$.

2.3 Task 2.3

Consider the following Python code:

```
import heapq

def addEdge(adj, u, v, w):
    adj[u].append((v, w))
    adj[v].append((u, w))
    return adj

def doingSomethingImportant(adj, V, src):
    pq = []
    heapq.heappush(pq, (0, src))
    cost = [float('inf')] * V
    cost[src] = 0

    while pq:
        d, u = heapq.heappop(pq)
        for v, weight in adj[u]:
            if cost[v] > cost[u] + weight:
                cost[v] = cost[u] + weight
                heapq.heappush(pq, (cost[v], v))

    return cost

V = 3
g = [[] for _ in range(V)]
g = addEdge(g, 0, 1, 4)
g = addEdge(g, 0, 2, 5)
g = addEdge(g, 1, 2, 4)

result = [
    (node, cost)
    for node, cost in enumerate(doingSomethingImportant(g, V, 0))
]

print(result)
```

Listing 3: Peak Finder 2D

Asymptotic Complexity: $O((E+V)\log V)$, where E is the number of edges and V is the number of vertices in the graph.

Explanation: The provided code demonstrates the implementation of Dijkstra's algorithm for finding the shortest paths from a source node to all other nodes in a weighted, undirected graph. The `addEdge` function is used to add edges to the graph, and `doingSomethingImportant` represents the main algorithm for calculating the minimum costs.
- The `doingSomethingImportant` function uses a priority queue (min-heap) to efficiently extract the minimum-cost node.
- The inner loop iterates through all the edges, and in the worst case, it can iterate through all edges and vertices, resulting in a complexity of $O((E + V) \log V)$.

3 Task 3

3.1 Our task:

The provided algorithm is designed to find a peak in a 2D matrix. A peak in this context is defined as a location where its four neighbors (north, south, east, and west) have values less than or equal to the value of the peak.

The `FindPeak` algorithm iterates through each element of the matrix, checking whether the current element is a peak by comparing it with its neighbors. The `IsValid` function ensures that the indices are within the matrix boundaries.

The algorithm returns the coordinates of the first peak found, or `None` if no peak is present in the matrix.

3.2 Time Complexity:

The time complexity of the given algorithm is O(row * log(column)). This is because the algorithm iterates through each element of the matrix, checking if it is a peak by comparing it with its neighbors. The loop iterates over all rows (O(row)), and within each row, the comparisons with neighbors have a constant time complexity.

The log(column) factor comes from the assumption that the columns are sorted or have some logarithmic structure. If the columns were not sorted or had no specific structure, the time complexity would be O(row * column) instead of O(row * log(column)).

3.3 Implemented Code:

```
    def find_peak(matrix):
        if not matrix:
            return None

        rows, columns = len(matrix), len(matrix[0])

        def binary_search(left, right):
            mid = (left + right) // 2
            max_in_column = -1
            max_index = -1

            for i in range(rows):
                if matrix[i][mid] > max_in_column:
                    max_in_column = matrix[i][mid]
                    max_index = i

            if mid > 0 and matrix[max_index][mid - 1] > matrix[max_index][mid
    ]:
                return binary_search(left, mid - 1)
            elif mid < columns - 1 and matrix[max_index][mid + 1] > matrix[
    max_index][mid]:
                return binary_search(mid + 1, right)
            else:
                return max_index, mid

        return binary_search(0, columns - 1)

    matrix1 = [
```

```
        [0, 1, 0],
        [1, 2, 1],
        [0, 1, 0]
    ]

    matrix2 = [
        [0, 1, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]

    matrix3 = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]

    matrix4 = [
        [5, 3, 7],
        [1, 9, 8],
        [6, 2, 4]
    ]

    print("Matrix 1:")
    print(matrix1)
    print("Peak Coordinates:", find_peak(matrix1))  # Output: (1, 1)

    print("\nMatrix 2:")
    print(matrix2)
    print("Peak Coordinates:", find_peak(matrix2))  # Output: (0, 1) or (2,
1)

    print("\nAdditional Test Case 1:")
    print(matrix3)
    print("Peak Coordinates:", find_peak(matrix3))  # Output: (2, 2) or any
peak in the corners

    print("\nAdditional Test Case 2:")
    print(matrix4)
    print("Peak Coordinates:", find_peak(matrix4))  # Output: (1, 1) or (1,
2) or (2, 1) or (2, 2)
```

Listing 4: Peak Finder 2D

3.4 Input

Example Usage:

```
matrix1 = [
    [0, 1, 0],
    [1, 2, 1],
    [0, 1, 0]
]

matrix2 = [
    [0, 1, 0],
    [0, 0, 0],
    [0, 0, 0]
```

```
]
```

Output:
`FindPeak(matrix1)` returns `(1, 1)`, and `FindPeak(matrix2)` returns `(0, 1)` or `(2, 1)`.
Additional Test Cases:

```
matrix3 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix4 = [
    [5, 3, 7],
    [1, 9, 8],
    [6, 2, 4]
]
```

Output:
`FindPeak(matrix3)` returns `(2, 2)` or any peak in the corners.
`FindPeak(matrix4)` returns `(1, 1)` or `(1, 2)` or `(2, 1)` or `(2, 2)`.