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

AEL Lab 0

Algorithm Engineering Lab

BSc in Software Engineering Program

Talimul Bari Shreshtho, 190042128

BSc in Software Engineering Program
Department of Computer Science and Engineering

Contents

1 Task 1

1.1 Given Functions

1. $f_1(n) = n^{0.9999} \cdot \log_2(n)$
2. $f_2(n) = 10^7 \cdot n$
3. $f_3(n) = \left(1000001 \cdot 10^{-6}\right)^n$
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 Functions Classes

By seeing the structure of the functions we can categorize them into different classes of functions.

1. $f_1(n) = n^{0.9999} \cdot \log_2(n)$ (Log-linear)
2. $f_2(n) = 10^7 \cdot n$ (Linear with a constant factor)
3. $f_3(n) = \left(1000001 \cdot 10^{-6}\right)^n$ (Exponential with a constant base)
4. $f_4(n) = n!$ (Factorial, highly non-linear)
5. $f_5(n) = 2^{10 \cdot 10^7}$ (Constant)
6. $f_6(n) = n \cdot \sqrt{n}$ (Polynomial)
7. $f_7(n) = n^{\sqrt{n}}$ (Superpolynomial)
8. $f_8(n) = \sum_{i=1}^{n}(i+1) \rightarrow \frac{n^2+3n}{2}$ (Quadratic)

1.3 Plot of the functions in Desmos

In the 1 we can see the plot of the functions. The graph is provided here.

1.4 Solution Task 1

1.4.1 Complexity of the functions

We can use the big o notation to understand the complexity of the functions. From the graphs we got from the Desmos after plotting them, we can see the complexity of the functions. The Y-axis on the graph represents the complexity of the functions and X-axis is the size of the input to the function.

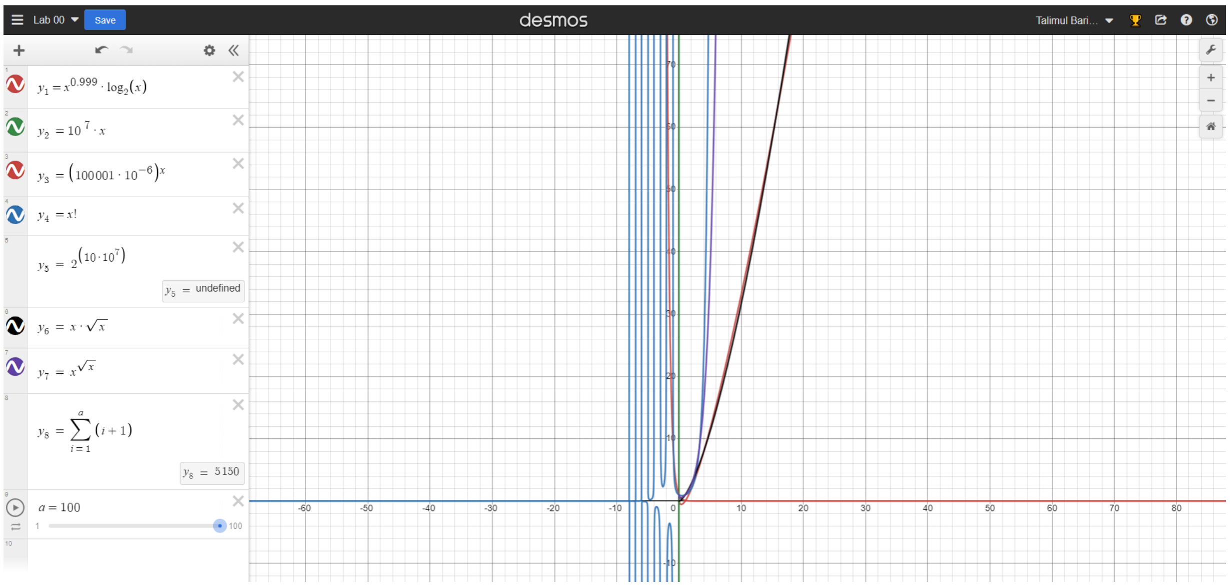


Figure 1: Time Complexity Representation of the functions in Desmos. The X-axis represents the input size and the y-axis is Time complexity

1.4.2 Functions sorted According to their Complexity

$f_5(n) < f_2(n) < f_1(n) < f_6(n) < f_8(n) < f_3(n)$(Because the base here is close to 1) $<$ $f_7(n) < f_4(n)$

Though the f_3 should have higher complexity than f_7 as it is a super polynomial and f_3 is an exponential function with the form of c^n the c here is close to 1. Its value is 1.000001 so the function has an almost linear complexity.

2 Task 2

2.1 Task 2.1

The given code is in 1. The code is of a function that finds a element of specific position in a array.

2.1.1 Analysis

- The initial if check, $j \geq \text{len}(a)$, is a constant-time operation ($O(1)$) as it involves a single comparison. If j is greater than or equal to the length of a, the function returns -1.
- The for loop responsible for iteration, $j < \text{len}(a)$, also results in constant time ($O(1)$). It returns the jth element on the first iteration. Therefore, the total complexity of the `findJthelement` function is $O(1)$.

Table 1: Functions and Their Complexity

Function	**Complexity**
$f_1(n) = n^{0.9999} \cdot \log_2(n)$	$O(n^{0.9999} \cdot \log_2(n))$
$f_2(n) = 10^7 \cdot n$	$O(n)$
$f_3(n) = \left(1000001 \cdot 10^{-6}\right)^n$	$O(c^n)$
$f_4(n) = n!$	$O(n!)$
$f_5(n) = 2^{10 \cdot 10^7}$	$O(1)$
$f_6(n) = n \cdot \sqrt{n}$	$O(n^{1.5})$
$f_7(n) = n^{\sqrt{n}}$	$O(n^{\sqrt{n}})$
$f_8(n) = \sum_{i=1}^{n}(i+1)$	$O(n^2)$

```
    def findJthelement(j, a):
        if j >= len(a):
            return -1
        for i in range(j, len(a)):
            return a[j]

    print(findJthelement(3, [1, 2, 3, 4, 5]))

```

Listing 1: Task 2.1 Code

2.2 Task 2.2

The given code is shown in code block 2

```
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: Task 2.2 Code

2.2.1 Algorithm Analysis

The function `doingSomethingImportant` in Code Snippet 2 updates a position of a 2D array, It takes a 2D array and a coordinate from the array as input and also the new value that replaces the old value after fetching the previous or current value from the array's coordinate. It selectively updates values equal to `prev` by traversing in four different directions:

1. The first two recursive calls keep the column fixed. The first call traverses upward, checking if the coordinates' values match the `prev` variable. If there's a match, it updates the values to the new value. The second call traverses downward for a similar check and update.

2. The last two recursive calls keep the row fixed. The third call traverses rightward, checking if the coordinate's value matches the `prev` value. If there's a match, it updates the value with the new value. The fourth call traverses leftward, replacing all matched values with `prev` with the new value.

So the **Worst Case Complexity :** $O(n \times m)$,

$$n = \text{number of rows} \quad (1)$$

$$m = \text{number of columns} \quad (2)$$

2.3 Task 2.3

The given code is shown in code block 3

```python
def build(a, v, tl, tr):
    if tl == tr:
        t[v] = a[tl]
    else:
        tm = (tl + tr) // 2
        build(a, v * 2, tl, tm)
        build(a, v * 2 + 1, tm + 1, tr)
        t[v] = t[v * 2] + t[v * 2 + 1]

a = [1, 2, 3, 4, 5, 6, 7, 8]
t = [0 for i in range(4 * len(a))]
build(a, 1, 0, len(a) - 1)
print(t)
```

Listing 3: The `build` Function

2.3.1 Complexity Analysis:

The `build` function is a recursive function that constructs a segment tree (`t`) from the given array (`a`). Here is the step-by-step analysis:

1. **Base Case:** If `tl` is equal to `tr`, it means a single element is left, and the value at position `v` in the segment tree is assigned the value of `a[tl]`.
2. **Recursive Calls:** If the base case is not met, the function calculates the middle index `tm = (tl + tr) // 2` and makes two recursive calls. One for the left child (`v * 2`) with the range `[tl, tm]`, and the other for the right child (`v * 2 + 1`) with the range `[tm + 1, tr]`.

Time Complexity: $O(n)$, where n = size of array `a`.

Note: The recurrence relation for time complexity is $T(n) = 2T(n/2) + O(1)$, which is a typical divide-and-conquer relation resulting in $O(n)$ complexity.

3 Task 3

3.1 Finding a Peak in a 2D Matrix

Given a 2D matrix, the goal is to find the coordinates of a peak. A peak 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.

3.1.1 Proposed Algorithm

The algorithm employs a binary search strategy to narrow down the potential peak column by column. The `is_peak` function checks if a point is a peak by comparing it with its neighbors. The `binary_search` function iteratively searches for a peak within a column.

Algorithm 1 Finding a Peak in a 2D Matrix

```
1: function FIND_PEAK(matrix)
2:     $rows \leftarrow$ length($matrix$)
3:     $columns \leftarrow$ length($matrix[0]$)
4:     function IS_PEAK($x, y$)
5:         $neighbors \leftarrow [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]$
6:         for $(nx, ny)$ in $neighbors$ do
7:             if $0 \leq nx < rows$ and $0 \leq ny < columns$ and $matrix[nx][ny] > matrix[x][y]$ then
8:                 return False
9:             end if
10:        end for
11:        return True
12:    end function
13:    function BINARY_SEARCH($left, right$)
14:        $mid \leftarrow (left + right) \div 2$
15:        $max\_in\_column \leftarrow$ max($matrix[i][mid]$ for $i$ in range($rows$))
16:        $max\_index \leftarrow$ index of $max\_in\_column$ in $matrix[max\_in\_column]$
17:        if $mid > 0$ and $matrix[max\_in\_column][mid - 1] > matrix[max\_in\_column][mid]$ then
18:            return BINARY_SEARCH($left, mid - 1$)
19:        else if $mid < columns - 1$ and $matrix[max\_in\_column][mid + 1] > matrix[max\_in\_column][mid]$ then
20:            return BINARY_SEARCH($mid + 1, right$)
21:        else
22:            return $(max\_in\_column, mid)$
23:        end if
24:    end function
25:    return BINARY_SEARCH($0, columns - 1$)
26: end function
```

3.1.2 Algorithm Implementation

```python
def find_peak(matrix):
    rows, columns = len(matrix), len(matrix[0])

    def is_peak(x, y):
        neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y +
 1)]
        for nx, ny in neighbors:
            if 0 <= nx < rows and 0 <= ny < columns and matrix[nx][ny] > matrix[x][y]:
                return False
        return True

    def binary_search(left, right):
        mid = (left + right) // 2
        max_in_column = max(matrix[i][mid] for i in range(rows))

        max_index = matrix[max_in_column].index(max_in_column)

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

    return binary_search(0, columns - 1)
```

Listing 4: Peak Finding Algorithm

3.1.3 Time Complexity

The time complexity is $O(\text{rows} \times \log(\text{columns}))$, where rows is the number of rows and columns is the number of columns in the matrix. This complexity arises from the binary search approach, which efficiently narrows down the potential peak column.

3.1.4 Test Cases

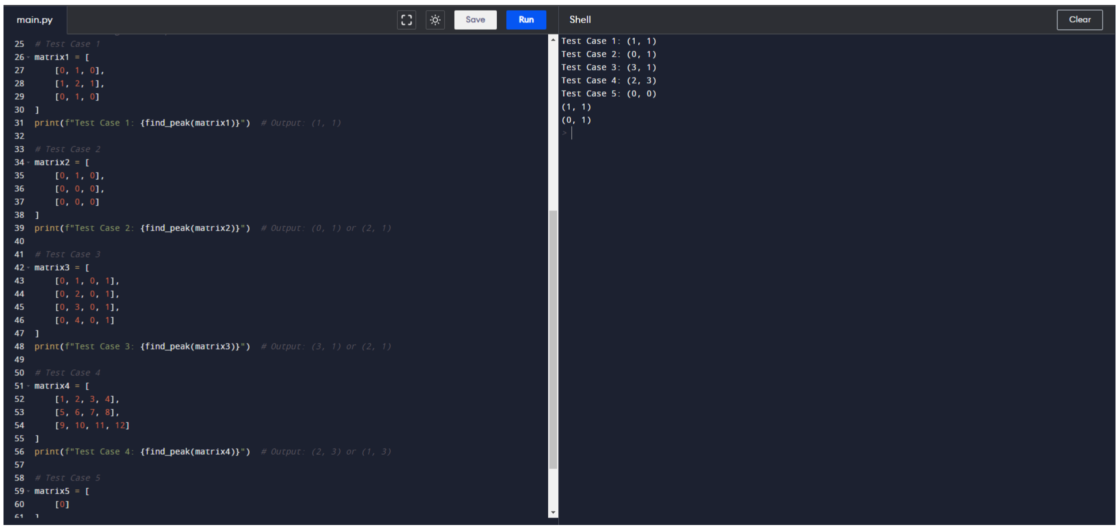
In figure 2 we run the implementation against some new test cases and check if the implemnetation outputs the correct answer not.

Test Case 1

Input Matrix:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Expected Output: (1, 1)
Actual Output: (1, 1)



```python
# Test Case 1
matrix1 = [
    [0, 1, 0],
    [1, 2, 1],
    [0, 1, 0]
]
print(f"Test Case 1: {find_peak(matrix1)}")  # Output: (1, 1)

# Test Case 2
matrix2 = [
    [0, 1, 0],
    [0, 0, 0],
    [0, 0, 0]
]
print(f"Test Case 2: {find_peak(matrix2)}")  # Output: (0, 1) or (2, 1)

# Test Case 3
matrix3 = [
    [0, 1, 0, 1],
    [0, 2, 0, 1],
    [0, 3, 0, 1],
    [0, 4, 0, 1]
]
print(f"Test Case 3: {find_peak(matrix3)}")  # Output: (3, 1) or (2, 1)

# Test Case 4
matrix4 = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
print(f"Test Case 4: {find_peak(matrix4)}")  # Output: (2, 3) or (1, 3)

# Test Case 5
matrix5 = [
    [0]
```

```
Test Case 1: (1, 1)
Test Case 2: (0, 1)
Test Case 3: (3, 1)
Test Case 4: (2, 3)
Test Case 5: (0, 0)
(1, 1)
(0, 1)
>
```

Figure 2: Testing the implementation

Test Case 2

Input Matrix:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Expected Output: (0, 1) or (2, 1)
Actual Output: (0, 1)

Test Case 3

Input Matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 3 & 0 & 1 \\ 0 & 4 & 0 & 1 \end{bmatrix}$$

Expected Output: (3, 1) or (2, 1)
Actual Output: (3, 1)

Test Case 4

Input Matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Expected Output: (2, 3) or (1, 3)
Actual Output: (2, 3)

Test Case 5

Input Matrix:

$$\begin{bmatrix} 0 \end{bmatrix}$$

Expected Output: (0, 0)
Actual Output: (0, 0)