Islamic University of Technology

CSE 4810

Algorithm Engineering Lab

Lab 0

Name: Tasnimul Hasnat

ID: 190041113

Section 1A

February 7, 2024

Contents

1 Task 1

1.1 Problem

Sort and draw the following functions with desmos in increasing order of asymptotic (big-O) complexity with justification:

a) $f_1(n) = n^{0.999}.\log_2(n)$

b) $f_2(n) = \binom{n}{2}$

c) $f_3(n) = (1000001 * 10^{-6})^n$

d) $f_4(n) = n!$

e) $f_5(n) = 2^{(10*10^7)}$

f) $f_6(n) = n * \sqrt{n}$

g) $f_7(n) = n^{\sqrt{n}}$

h) $f_8(n) = \sum_{i=1}^{n}(i + 1)$

1.2 Graph

Plotting the given functions in desmos graphing calculator we get -

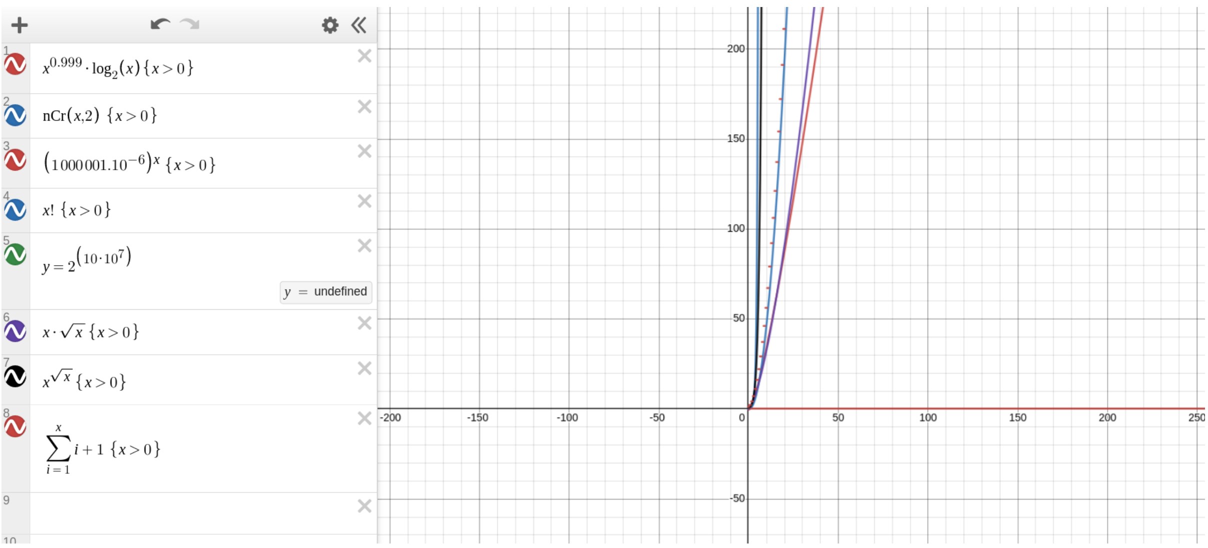

Figure 1: Plotted Function in Desmos

N.B - f_5 is a large value (2^{10^8}) so it didn't fit inside the graph.

1.3 Solution

The complexities of the given equations are -

a) $f_1(n) = n^{0.999}.\log_2(n)$
   For any c >0, $\log_2(n)$ is $O(n^c)$.
   So, $n^{0.999}.\log_2(n) = O(n^{0.999}.n^{0.001}) = O(n)$
   Complexity = $O(n)$ or linear

b) $f_2(n) = \binom{n}{2}$
   $\binom{n}{2} = \frac{n(n-1)}{2}$
   Complexity = $O(n^2)$ or polynomial

c) $f_3(n) = (1000001 * 10^{-6})^n$
   $(1000001 * 10^{-6})^n = 1.000001^n$
   Complexity = $O(1.000001^n)$ or exponential

d) $f_4(n) = n!$
   Complexity = $O(n!)$ or factorial

e) $f_5(n) = 2^{(10*10^7)}$
   Complexity = $O(1)$ or constant

f) $f_6(n) = n * \sqrt{n}$
   Complexity = $O(n^{3/2})$ or polynomial

g) $f_7(n) = n^{\sqrt{n}}$
   $n^{\sqrt{n}} = (2^{logn})^{\sqrt{n}} = 2^{\sqrt{n}logn}$
   Complexity = $O(2^{\sqrt{n}logn})$

h) $f_8(n) = \sum_{i=1}^{n}(i+1)$
   $\sum_{i=1}^{n}(i+1) = \frac{n^2+n}{2} + n = \frac{n^2+3n}{2}$
   Complexity = $O(n^2)$ or polynomial

1.4 Complexity Comparison

So, the increasing order of complexity –

Complexity Comparison				
Rank	**Function**	**Values**	**Complexity**	$O(n)$
1	$f_5(n)$	$2^{(10*10^7)}$	Constant	$O(1)$
2	$f_1(n)$	$n^{0.999}.\log_2(n)$	Linear	$O(n)$
3	$f_6(n)$	$n * \sqrt{n}$	Polynomial	$O(n^{1.5})$
4	$f_2(n)$	$\binom{n}{2}$	Polynomial	$O(n^2)$
5	$f_8(n)$	$\sum_{i=1}^{n}(i+1)$	Quadratic	$O(n^2)$
6	$f_7(n)$	$(n)^{\sqrt{n}}$	Exponential	$O(2^{\sqrt{n}logn})$
7	$f_3(n)$	$(1000001 * 10^{-6})^n$	Exponential	$O(1.000001^n)$
8	$f_4(n)$	$n!$	Factorial	$O(n!)$

2 Task 2

2.1 Find only the complexity for find3RDelement function

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

The function **find3RDelement** searches for the 3rd element in a list. Whatever the length of the list is in the 3rd iteration the loop will return the the value. Thus the **find3RDelement** is constant or $\mathbf{O(1)}$.

2.2 Find only the complexity for doingSomethingImportant

```
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)
```

The **doingSomethingImportant** function is a flood fill algorithm. The time complexity of the algorithm depends on the size of the array. Thus for an array with n rows and m columns, the complexity will be $O(nm)$.

2.3 Find only the complexity for doingSomethingImportant

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

The **doingSomethingImportant** function is the Dijkstra algorithm to find the shortest path from one node to another. The implementation uses a priority queue/min-heap. Thus the complexity will be $O((V + E)\ logV)$.

3 Task 3

3.1 Solution

2D Peak - Binary Search (Recursive)

Steps:

a) Pick middle column j = m/2

b) Find global maximum on column j at (i, j)

c) Compare (i, j − 1), (i, j), (i, j + 1)

d) Pick left columns if (i, j − 1) > (i, j) , Otherwise pick right columns

e) (i, j) is a 2D-peak if neither condition holds

f) Solve the new problem with half the number of columns.

g) When you have a single column, find global maximum and youre done.

Whenever the domain is reduced to a single column, the coordinate having the max value of that column is its 2D peak.

3.2 Algorithm

```
def peakFinder2D_binary(matrix):

    rows, cols = matrix.shape

    if cols == 1:  # Base case
        return np.argmax(matrix[:, 0]), 0

    mid_col = cols // 2
    max_row_index = np.argmax(matrix[:, mid_col])
    max_val = matrix[max_row_index, mid_col]

    if mid_col > 0 and matrix[max_row_index, mid_col - 1] > max_val:
        # Recurse on left half
        return peakFinder2D_binary(matrix[:, :mid_col])

    elif mid_col < cols - 1 and matrix[max_row_index, mid_col + 1] >
    ↪   max_val:
```

```
        # Recurse on right half
        return peakFinder2D_binary(matrix[:, mid_col + 1:])

    else:
        return max_row_index, mid_col  # Found a peak
```

3.3 Complexity

- **Divide-and-Conquer Approach :** The function recursively divides the matrix in half, effectively halving the search space in each iteration. This logarithmic behavior leads to the **log(m)** component in the complexity.
- **Finding Maximum in a Column:** The argmax operation used to find the maximum element in a column takes **O(n)** time, as it needs to iterate through all elements in that column. This contributes to the n component in the complexity.
- **Overall complexity :** Combining these two factors, we find the total complexity as $\mathbf{O}(n * log_2(m))$

In every iteration we are reducing the search domain by half in terms of columns. So the **Complexity** $= \mathbf{O}(n * log_2(m))$ where n and m is the number of row and columns respectively.