Roll No: EE18B067 Name:Abhishek Sekar

Date: October 8, 2021

1. Order the functions by asymptotic growth rate.

| | | |
|---|---|---|
| 4nlog(n) + 2n | $2^{10}$ | $2^{log(n)}$ |
| 3n + 100log(n) | 4n | $2^n$ |
| $n^2$ + 10n | $n^3$ | nlog(n) |

**Solution:**
**Big O notation**
Making use of the Big O notation,we can write the order of each of these functions.

| Function | Order |
|---|---|
| 4nlog(n) + 2n | $\mathcal{O}(n\log(n))$ |
| $2^{10}$ | $\mathcal{O}(1)$ |
| $2^{log(n)}$ | $\mathcal{O}(n^{(log2)})$ |
| 3n+100log(n) | $\mathcal{O}$(n) |
| 4n | O(n) |
| $2^n$ | $\mathcal{O}(2^n)$ |
| $n^2$ + 10n | $\mathcal{O}(n^2)$ |
| $n^3$ | $\mathcal{O}(n^3)$ |
| nlog(n) | $\mathcal{O}(n\log(n))$ |

Having, written this, we can now easily order the functions by asymptotic growth rate. For the cases where the order is the same, we see that 4nlog(n) +2n > nlog(n) and for the $\mathcal{O}$(n) terms, we see that 4n > 3n + 100log(n) (as the additional n has a higher asymptotic growth than log(n)) .
Therefore, the order of asymptotic growth rate for these functions is, $2^n > n^3 > n^2 + 10n >$ 4nlog(n) + 2n $\geq$ nlog(n) $> 4n \geq$ 3n + 100log(n) $> 2^{log(n)} > 2^{10}$. (as log(2) is less than 1)

2. In each of the following situations, indicate whether, f = $\mathcal{O}$(g) or f = $\Omega$(g), or both,(in which case f = $\Theta$(g)). Justify your answer.

| | f(n) | g(n) |
|---|---|---|
| (i) | n-100 | n-200 |
| (ii) | log(2n) | log(3n) |
| (iii) | $n^{0.1}$ | $log(n)^{10}$ |
| (iv) | n$2^n$ | $3^n$ |

**Solution:**
**Some definitions:**

$$T(n) = \mathcal{O}(f(n)) : \exists\ c > 0, \exists\ n_0 > 0, \text{S.T } \forall n \geq n_0, T(n) \leq c \cdot f(n) \quad (1)$$
$$T(n) = \Omega(f(n)) : \exists\ c > 0, \exists\ n_0 > 0, \text{S.T } \forall n \geq n_0, T(n) \geq c \cdot f(n) \quad (2)$$
$$T(n) = \Theta(f(n)) : \text{if } T(n) = \mathcal{O}(f(n)), T(n) = \Omega(f(n)) \quad (3)$$

Using these definitions, we can attempt to solve the problem.

- (i): Let us check if f(n) is $\mathcal{O}(g(n))$ here. By the definition, we need to show that there exists a positive $n_0$, above which f(n) is bounded above by c·g(n) for some positive c. Let us take c = 2, then, if f(n) is $\mathcal{O}(g(n))$, we have,

$$\begin{aligned} n - 100 &\leq 2 \cdot (n - 200) \\ n - 100 &\leq 2n - 400 \\ n &\geq 300 \end{aligned}$$

We've shown that there exists an $n_0 = 300$. Therefore, f(n) is $\mathcal{O}(g(n))$.
Let us check if f(n) is $\Omega(g(n))$ here. By the definition, we need to show that there exists a positive $n_0$, above which f(n) is bounded below by c·g(n) for some positive c. Let us take c = 0.5, then, if f(n) is $\Omega(g(n))$, we have,

$$\begin{aligned} n - 100 &\geq 0.5 \cdot (n - 200) \\ n - 100 &\geq 0.5n - 100 \\ n &\geq 0.5n \end{aligned}$$

Which is true for all n. Therefore, we can take our $n_0$ to be any positive number. Therefore, f(n) is $\Omega(g(n))$ as well.
As f(n) is both $\mathcal{O}(g(n))$ and $\Omega(g(n))$, f(n) is $\Theta(g(n))$.
In the below plot, we can see that this is indeed the case as f(n) is nestled between the two functions we considered above.

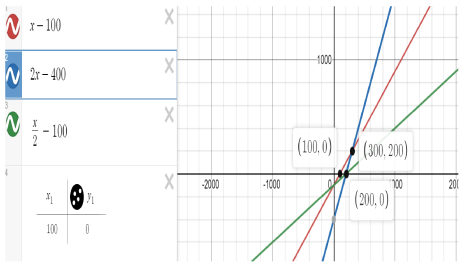


Figure 1: Visualisation of $f(n) = \Theta(g(n))$

- (ii) Proceeding as we did in the previous subdivision, let us check if f(n) is $\mathcal{O}(g(n))$. We take c = 1, then if f(n) is $\mathcal{O}(g(n))$, we have,

$$\begin{aligned} &log(2n) \leq c \cdot log(3n) \\ &log(2n) \leq log(3n) \\ &2n \leq 3n \qquad \text{(Taking anti-log)} \end{aligned}$$

Which is true for all n. Therefore, we can consider any positive $n_0$ and f(n) is $\mathcal{O}(g(n))$.
Let us check if f(n) is $\Omega(g(n))$. Consider c = 0.5, then if the condition holds true, we have,

$$\begin{aligned} &log(2n) \geq c \cdot log(3n) \\ &log(2n) \geq 0.5log(3n) \\ &2n \geq 3n^{0.5} \qquad \text{(Taking anti-log)} \\ &2n \geq \sqrt{3n} \\ &\sqrt{n} \geq \frac{\sqrt{3}}{2} \\ &n \geq \frac{3}{4} \end{aligned}$$

Therefore, we've shown that there exists an $n_0 = \frac{3}{4}$ and f(n) is $\Omega(g(n))$.
As f(n) is both $\mathcal{O}(g(n))$ and $\Omega(g(n))$, f(n) is $\Theta(g(n))$.
In the below plot, we can see that this is indeed the case as f(n) is nestled between the two functions we considered above.

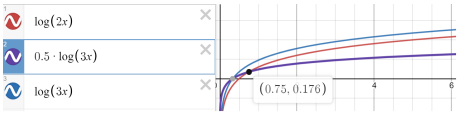


Figure 2: Visualisation of $f(n) = \Theta(g(n))$

- (iii) Let us see if f(n) is $\mathcal{O}(g(n))$. If this were to be true, for appropriate c and $n_0$, we'll have,

$$\begin{aligned} & f(n) \leq c \cdot g(n) \\ & \frac{f(n)}{g(n)} \leq c && \forall n \geq n_0 \\ & \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c \end{aligned}$$

  Likewise, we can argue in a similar manner for the f(n) is $\Omega(g(n))$ case and we get,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$

  Utilizing these to the given problem, we see that,

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^{0.1}}{log(n)^{10}} \\ &\Rightarrow \lim_{n\to\infty} \frac{0.1}{10} \cdot \frac{n^{-0.9}}{log(n)^9 \cdot \frac{1}{n}} && \text{(Using L'Hospital's rule)} \\ &\Rightarrow \lim_{n\to\infty} \frac{0.1}{10} \cdot \frac{n^{0.1}}{log(n)^9} \end{aligned}$$

  On repeatedly implementing L'Hospital's rule, we get

$$\begin{aligned} &\Rightarrow \lim_{n\to\infty} \frac{0.1}{10} \cdot \frac{0.1}{9} \cdots \frac{0.1}{1} n^{0.1} \\ &\Rightarrow \lim_{n\to\infty} \frac{(0.1)^{10}}{10!} \cdot n^{0.1} = \infty \end{aligned}$$

  From this, we see that $\infty > c$ for the chosen c and therefore, f(n) is $\Omega(g(n))$.We see that f(n) is not $\mathcal{O}(g(n))$ as $\infty < c$ is not true for finite c.

- (iv) Proceeding as we did in the previous problem, we see that,

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n2^n}{3^n} \\ &\Rightarrow \lim_{n\to\infty} \frac{n}{\frac{3}{2}^n} \\ &\Rightarrow \lim_{n\to\infty} \frac{1}{\frac{3}{2}^n log\left(\frac{3}{2}\right)} = 0 && \text{(Using L'Hospital's rule)} \end{aligned}$$

  From this, we see that $c > 0$ for the chosen c and therefore, f(n) is $\mathcal{O}(g(n))$. f(n) is not $\Omega(g(n))$ as $c < 0$ is not true as c is positive.

3. Describe an efficient algorithm for finding the ten largest elements in a sequence of size n. What is the running time of your algorithm?

**Solution:**
**Code:**

```
def ten_largest(arr):
    if(len(arr)-10<10):
        for i in range(len(arr)-10):
            min_index = i
            for j in range(i+1,len(arr)):
                if(arr[j]<arr[min_index]):
                    min_index = j

            arr[i],arr[min_index] = arr[min_index],arr[i]

    else:
        for i in range(10):
            for j in range(len(arr)-i-1):
                if(arr[j] > arr[j+1]):
                    arr[j],arr[j+1] = arr[j+1],arr[j]

    return arr[-10:]
```

**Input and Output**

- **Input:** Integer array arr.
- **Output:** An ordered/unordered array containing the ten largest elements in arr.

**Sample Input and Sample Output:**

- input: [0, 2, 1, 9, 8, 9, 7, 6, 5, 2, 1, 3, 4, 4, 5, 6, 7, 8, 23, 21, 24] output: [6, 7, 7, 8, 8, 9, 9, 21, 23, 24]
- input:[0, 2, 1, 9, 8, 9, 7, 6, 5, 2, 1, 3, 4, 4, 5, 6, 7, 8] output: [5, 9, 8, 9, 7, 6, 5, 6, 7, 8]

**Explanation**
The function first checks the length of the array.
If it sees that the length of the array is less than 20, it tries computing the first n-10 smallest values in the array where n is it's length. This process is not as expensive as computing the 10 largest values as in this case $10 > n - 10$. At the $i^{th}$ outer iteration,the array is parsed from index i+1 till its end and the $i^{th}$ smallest value present in the array is found and moved to the $i^{th}$ index.
If the length of the array is larger than 20, then the function iterates through the array 10 times. At the $i^{th}$ outer iteration, the array is parsed till the $n - i - 1$ index and in this process, the $i^{th}$ largest value present in the array is computed.This large value sinks to the $i^{th}$ index from the end of the array on completion of the iteration.
Finally, in either case, the last 10 values host the 10 largest elements of the array and hence that is returned from the function.
**Time Complexity:**
As we iterate through 0 to 9 in the outer for loop and from 0 to n-i-1 in the inner for loop (for larger sized arrays), we perform $\mathcal{O}(10n)$ computations for the worst case.
Therefore, the time complexity of this algorithm is $\mathcal{O}(n)$.

4. Use the divide and conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

**Solution:**
**Code:**

```
def extend_length(A,B):
    len_a = len(A)
    len_b = len(B)
    if (len_a > len_b):
        B = '0'*(len_a - len_b) + B
    elif (len_b > len_a):
        A = '0'*(len_b - len_a) + A
    return A,B

def add(A, B):
    if(len(A) != len(B)):
        A,B = extend_length(A,B)
    sum_AB = ""
    carry_AB = 0
    for i in range(len(A) - 1,-1,-1):
        a,b = int(A[i]),int(B[i])

        #using full adder logic
        s    = (a ^ b) ^ carry_AB
        sum_AB = str(s) + sum_AB
        carry_AB = (a & b) | (a & carry_AB) | (b & carry_AB)

    if(carry_AB): #overflow
        sum_AB = '1'+ sum_AB
    return sum_AB

def multiply(A, B):

    if(len(A) != len(B)):
        A,B = extend_length(A,B)

    n = len(A)

    #checking for boundary cases
```

```
    if (n == 0):
        return 0
    if (n == 1):
        return int(A)&int(B)

  #split A into Al and Ar right at the "middle"
    A_l , A_r = A[:n//2],A[n//2:]
    B_l , B_r = B[:n//2],B[n//2:]



    P1 = multiply(A_l, B_l) #multiplication of the MSB side
    P2 = multiply(A_r, B_r) #multiplication of the LSB side
    P3 = multiply(add(A_l, A_r), add(B_l, B_r)) #for cross terms

    print('Karatsuba algorithm in action:')
    print (P1,' x ' ,(1 << 2*(n-n//2)), ' + ', ' ( ',P3,' - ',P1,' - ',P2,' ) x ',
    (1 << (n-n//2)), ' + ', P2)

    return P1 * (1 << 2*(n-n//2)) + (P3-P1-P2)*(1 << (n-n//2)) + P2

A = "10011011"
B = "10111010"

print("The answer is" ,bin(multiply(A,B))[2:],'or in decimal,',multiply(A,B))
```

---

**Explanation**

The algorithm that has been implemented in this problem is the Karatsuba divide and conquer multiplication algorithm which has been taught in the class notes provided.

In a nutshell,a naive divide and conquer multiplication of two n bit numbers A and B would be as follows:

$$A \cdot B = (A_l \cdot B_l) \cdot 2^n + (A_l \cdot B_r + B_l \cdot A_r) \cdot 2^{n/2} + A_r \cdot B_r$$

where $A_l, A_r$ refers to the left and right halves of the binary number A. But this requires 4 multiplication steps for every recursion. Here's where Karatsuba's idea comes in. $(A_l \cdot B_r + B_l \cdot A_r)$ is computed as $((A_l + A_r) \cdot (B_r + B_l) - A_l \cdot B_l - A_r \cdot B_r)$. Therefore, here we just need to compute three values instead of the above 4.

The function *extend length* extends the length of a number by adding 0s at the front to ensure both the numbers are equally long.

The function *add* performs a binary addition which works like a ripple carry adder (i.e., a series of concatenated Full Adders).

Finally, the function *multiply* performs Karatsuba's multiplication algorithm described above and this results in the product of the numbers 10011011 and 10111010 being 111000010011110 or in decimal, 28830.

**Time Complexity:**

We perform 3 multiplications every function call. Let the time complexity of the algorithm be O. Then we have,the dominating term of the time complexity as,

$$O(n) = 3 \cdot O(n/2)$$

where O(n) represents the time complexity of multiplying 2 n-bit numbers. On solving the recursion, we get the net time complexity of the algorithm to be $\mathcal{O}\left(n^{\frac{log3}{log2}}\right)$ or approximately, $\mathcal{O}\left(n^{1.6}\right)$

5. You are given a unimodal array of n distinct elements, meaning that its entries are in increasing order up until its maximum elements, after which its elements are in decreasing order. Give an algorithm to compute the maximum element of a unimodal array that runs in $\mathcal{O}$(log(n)) time.

**Solution:**
**Code:**

```
def find_max(arr, left , right):
    if (right>=left):
        mid = (left + right)//2
        if ((arr[mid] > arr[mid +1]) and (arr[mid] > arr[mid -1])):
            return arr[mid]
        elif ((arr[mid] > arr[mid +1]) and (arr[mid] < arr[mid -1])):
            right = mid -1
            return find_max(arr, left, right)
        elif ((arr[mid] < arr[mid +1]) and (arr[mid] > arr[mid -1])):
            left = mid +1
            return find_max(arr, left, right)
    else: #error for a non-unimodal array
        return -1

find_max(arr,0,N-1)
```

**Input and Output**

- **Input:** Integer array arr.
- **Output:** maximum valued element of arr.

**Sample Input and Sample Output:**

input: [1, 2, 3, 4, 3, 2, 1] output: 4
**Explanation**
The function utilizes a divide and conquer approach to solve this problem. It first computes the midpoint of the input array (at that particular recursive call) and then checks if the midpoint hosts the maximum element(as the array is unimodal, only the maximum element will be larger than elements on either side). If it does, then it returns the maximum element. If it doesn't it checks if the midpoint lies on the increasing or the decreasing side of the array and correspondingly repeats the said approach on the decreasing and the increasing side of the array (originating or terminating near the midpoint) respectively.
**Time Complexity:**
As we iterate through the array in a divide and conquer fashion, the time complexity of this algorithm is $\mathcal{O}$(logn) where n is the length of the array.

6. [Towers of Hanoi]
Given a game board with three pegs and a set of disks of different diameter all stacked from smallest to largest on the leftmost

peg, moves all of the disks to the rightmost peg following these two rules. First, only one disk may be moved at a time. Second, a larger diameter disk may never be placed on a smaller disk. Any number of disks can be used. Implement this in Python.

**Solution:**
**Code:**

```
def Towers_of_Hanoi(n , source_peg, dest_peg,inter_peg ):
    if (n==1):
        print ("Move disk number 1 from",source_peg,"to",dest_peg)
        return
    Towers_of_Hanoi(n-1, source_peg, inter_peg, dest_peg)
    print ("Move disk number",n,"from",source_peg,"to",dest_peg)
    Towers_of_Hanoi(n-1, inter_peg, dest_peg, source_peg)
```

**Input and Output**

- **Input:** Integer N.
- **Output:** Sequence of moves to win the game.

**Sample Input and Sample Output:** input : 3
output:
Move disk number 1 from leftmost peg to rightmost peg
Move disk number 2 from leftmost peg to middle peg
Move disk number 1 from rightmost peg to middle peg
Move disk number 3 from leftmost peg to rightmost peg
Move disk number 1 from middle peg to leftmost peg
Move disk number 2 from middle peg to rightmost peg
Move disk number 1 from leftmost peg to rightmost peg
**Explanation**

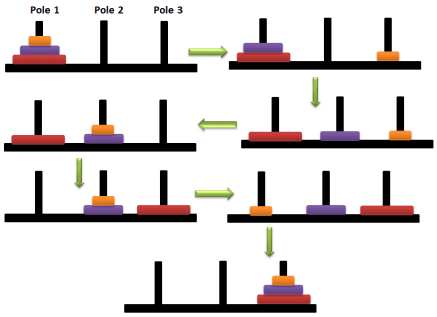


Figure 3: Towers of Hanoi for n=3

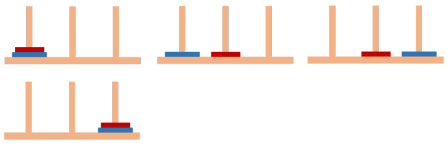

Figure 4: Towers of Hanoi for n=2

Based on the rules of the game, we see that we can devise a divide and conquer algorithm for this problem. The goal of the game is to shift all the disks from the leftmost peg to the rightmost one complying to the rules. We see that the methodology to win the game is as follows for n discs. Stack the n-1 smallest discs on the middle peg shift the largest disc to the rightmost peg. From these n-1 smallest discs, n-2 of them have to be stacked back on to the leftmost peg so as to guide the $(n-1)^{th}$ largest disc onto the rightmost peg. After this the problem just reduces to a Towers of Hanoi problem but with n-2 kegs this is what has been done in the divide and conquer algorithm. Moreover, this stacking of n-1 discs in the intermediary step can be treated as an equivalent Towers of Hanoi game where the destination, intermediary and the source pegs are different.

**Time Complexity:**

At each step of the Towers of Hanoi problem, there are two function calls to a smaller Towers of Hanoi problem. This gives rise to the dominating term in the time complexity as follows. If O(n) is the time complexity of the Towers of Hanoi problem with n discs, we have,

$$O(n) = 2 \cdot O(n-1)$$

Therefore,solving this recursion, we get the time complexity of this algorithm as $\mathcal{O}(2^n)$.