Çağrı Yıldız

1901042630

CSE 222 Homework-2

Q1 :

a-) $f(n) = (n^2 - 3n)^2$ and $g(n) = 5n^3 + n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{(n^2-3n)^2}{5n^3+n} = \frac{n^4 - 6n^3 + 9n^2}{5n^3+n} = \frac{\infty}{\infty}$$

solution 1:

$$\frac{a_n x^n}{b_m x^m} \Rightarrow \begin{cases} n < m \Rightarrow 0 \\ n = m \Rightarrow \frac{a}{b} \\ n > m \Rightarrow \pm\infty \end{cases}$$

so $n > m$; therefore $f(n) \in \Omega(g(n))$

solution 2: $\xrightarrow{\text{L'Hospital}} \frac{n^4 - 6n^3 + 9n^2}{5n^3+n} \longrightarrow \frac{4n^3 - 18n^2 + 18n}{15n^2 + 1} \xrightarrow{\text{L'Hospital}} \frac{12n^2 - 36n + 18}{30n} \xrightarrow{\text{L'Hospital}} \frac{24n - 36}{30} = \infty$

Therefore $f(n) \in \Omega(g(n))$

---

b-) $f(n) = n^3$ and $g(n) = \log_2 n^4$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^3}{\log_2 n^4} = \frac{n^3}{4\log_2 n} = \frac{\infty}{\infty} \xrightarrow{\text{L'Hospital}} \frac{3n^2}{\frac{4}{n \cdot \ln 2}} \longrightarrow \frac{3n^2 \cdot n \cdot \ln 2}{4} \rightarrow \frac{3n^3 \cdot \ln 2}{4} = \infty$$

Therefore $f(n) \in \Omega(g(n))$

---

c-) $f(n) = 5n \cdot \log_2(4n)$ and $g(n) = n \cdot \log_2(5^n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{5n \cdot \log_2(4n)}{n \cdot \log_2(5^n)} = \frac{5n \cdot (\log_2 4 + \log_2 n)}{n^2 \cdot \log_2 5} = \frac{5\cancel{n}(2 + \log_2 n)}{n^{\cancel{2}} \cdot \log_2 5} = \frac{5(2+\log_2 n)}{n \cdot \log_2 5}$$

$$\rightarrow \frac{10 + 5\log_2 n}{n \cdot \log_2 5} \rightarrow \frac{\log_2 n}{n} \rightarrow$$ logarithmic growth is slower than polynomial growth

$$\lim_{n\to\infty} \frac{\log_2 n}{n} = 0 \longrightarrow$$ Therefore $f(n) \in O(g(n))$

d-) $f(n) = n^n$ and $g(n) = 10^n$

$\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \frac{n^n}{10^n} \longrightarrow \frac{\infty}{\infty} \longrightarrow$ the n^n growth is faster than a^x when we have a $\frac{\infty}{\infty}$ limit condition.

solution 1:

! $x^x > x! > a^x > x^a > \log_a x > \begin{matrix} \sin ax \\ \cos ax \end{matrix}$

$\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Therefore $f(n) \in \Omega(g(n))$

solution 2: $\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \frac{n^n}{10^n} = \frac{\cancel{n} . \ln(n)}{\cancel{n} . \ln 10} \rightarrow \lim\limits_{n \to \infty} \frac{\ln(n)}{\ln 10} \longrightarrow \infty$

---

e-) $f(n) = 8n . \sqrt[5]{2n}$ and $g(n) = n . \sqrt[3]{n}$

$\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \frac{8n . \sqrt[5]{2n}}{n . \sqrt[3]{n}} \longrightarrow \frac{8n . (2n)^{\frac{1}{5}}}{n . n^{\frac{1}{3}}} \longrightarrow \frac{16 . n^{\frac{6}{5}}}{n^{\frac{4}{3}}} \longrightarrow 16n^{\frac{-2}{15}}$

Therefore $f(n) \in O(g(n))$ $\longleftarrow$ $\lim\limits_{n \to \infty} \frac{16}{n^{\frac{2}{15}}} = 0$

Q_2 :

a-)
```
static void methodA (String str_array[]){
    for (int i = 0; i < str_array.lenght ; i++)
        str_array[i] = " ";
}
```

worst-case scenario

$$\left.\begin{array}{l} str\_array[0] = " " \\ str\_array[1] = " " \\ \vdots \\ str\_array[n-1] = " " \end{array}\right\} n \text{ times}$$

* The method iterates through each element of the input array 'str_array', which has n elements.

* For each operation of the loop (from 0 to n-1), it performs a constant time operation: assigning an empty string (" ") to current element of the array.

* Since the loops run n times and each operation within the loop is considered to constant time, the total time complexity of this method is a lineer function of n.

⟶ Worst-case time complexity of methodA is $\underline{O(n)}$.

---

b-)
```
static void methodB ( String str_array[] ){
    for(int i = 0 ; i < str_array.lenght ; i++)
        methodA(str_array);
    for( int j = 0 ; j < str_array.lenght ; j++)
        System.out.println (str_array[i]);
}
```

worst-case scenario

$$\left.\begin{array}{l} i=0 \rightarrow methodA \\ i=1 \rightarrow methodA \\ \vdots \\ i=n-1 \rightarrow methodA \end{array}\right\} \text{calling n times } O(n) \text{ complexity function } O(n).O(n) = O(n^2)$$

$$\left.\begin{array}{l} i=0 \rightarrow print \\ i=1 \rightarrow print \\ \vdots \\ i=n-1 \rightarrow print \end{array}\right\} n \text{ time prints}$$

* The first 'for' loop runs n time, where n is the lenght of 'str_array'. Each iteration calls 'methodA (str_array)', which has a time complexity of $O(n)$. Therefore, the total complexity contributed by this loop is $O(n) * O(n) = O(n^2)$

* The second 'for' loop also runs n times, which is considered a constant time operation, $O(1)$, Hence, the total complexity of this loop is $O(n)$

⟶ Therefore the methodB time $O(n^2) + O(n)$. So the worst case time complexity of methodB is $\underline{O(n^2)}$

c-)
```
static void methodC (String str_array[]) {
    for(int i=0; i < str_array.lenght; i++)
        for( int j =0 ; j < str_array.lenght; j++)
            methodB(str_array);
}
```
⟶ $O(n^2)$ time complexity

! worst-case scenario

(j=0) methodB
(j=1) methodB
⋮
(j=n-1) methodB

n times calling methodB

i=0
i=1
i=n-1

n times calling other loop
so $n \cdot n \cdot n^2 = O(n^4)$

* methodC consist of a nested loop structure where each loop iterates n times, where n is the lenght of str_array

* For each iteration of the inner loop methodB f. is called. Since there are n iterations of the outer loop and n iteration of the inner loop, methodB is called $n \times n = n^2$ times

* Given that methodB itself has a time complexity of $O(n^2)$, the total time complexity contributed by all calls to methodB within methodC will be $O(n^2) \times O(n^2) = O(n^4)$ in worst case scenario.

---

d-)
```
static void methodD (String str_array[]){
    for (int i = 0; i < str_array.lenght; i++){
        System.out.println (str_array[i]);
        str_array[i--] = "";
    }
}
```
worst-case scenario

i=0
print
i=-1
i=0

infinite loop

* the loop is intended to iterate through each element of the array 'str_array', which has a lenght of n

* Within the loop, it prints the current element to the console, which is a constant-time operation $O(1)$

* Then, it assigns an empty string " " to the current element, which is also a constant time operation $O(1)$.

* However, after the updating current element, the loop counter 'i' is decremented by 1 with 'i--'. This means that the loop will revisit the same index on the next iteration, resulting in an infinite loop if it were not for the modification of the array element that affects the loops termination condition. So we can't say any worst case time complexity. Maybe $O(\infty)$ in theory.

e-)

```
static void methodE (String str_array []) {
    for (int i = 0; i < str_array.lenght; i++)
        if (str_array [i] == "")
            break;
}
```

in worst case scenario

$$\left.\begin{array}{ll} i=0 & arr[0] \\ i=1 & arr[1] \\ \vdots & \\ i=n-1 & arr[n-1] \end{array}\right\} \text{checks n time to array}$$

* The loop iterates through each element of the input array 'str_array', which has a lenght of n.

* For each iteration, it checks if the current element equals an empty string "". If so, it breaks out of the loop; othorwise, it continues to the next iteration.

* The worst-case time complexity of this method is O(n). This is because, in the worst case, the loop must check each element of the array once before terminating.

Q_3:

a-) Array is Sorted in Ascending Order

* pseudo-code:

Algorithm FindMaxDifferenceSorted (A)

Input: An array A of lenght n, sorted in ascending order.

Output: The maximum difference between 2 elements in A

```
1. If n <= 1 then
2.     return 0    // No difference can be calculated
3. End If
4. maxDiff = A[n-1] - A[0]   // A[n-1] is highest, A[0] is lowest
5. return maxDiff
```

Array is sorted so, to find max difference we substract the last element from first one. The algorithm directly accesses the first and last elements of the array and calculates the difference. So, the time complexity of this algorithm is O(1), as it performs a constant number of operations, irrespective of the size of the input array.

b-) Array is Not Sorted

* pseudo-code:

```
Algorithm FindMaxDifferenceUnsorted (A)
Input: An array A of lenght n
Output: The maximum difference between two elements in A
If n <= 1 then // checking the initial condition
    return 0 // No difference can be calculated
End If
Initialize minArr = A[0] // Initializing the variable for min element
Initialize maxArr = A[0] // Initializing the variable for max element
For i = 1 to n-1 do // for loop for finding min/max elements in array
    If A[i] < minArr then // Condition for min element
        minArr = A[i]
    Else If A[i] > maxArr then // Condition for max element
        maxArr = A[i]
    End If
End for
maxDiff = maxArr - minArr // Finding the maximum difference
return maxDiff
```

* The algorithm initializes the minArr and maxArr with the value of the first element in the array. It then through the array starting from the second element. In the loop we have conditions for finding the min and max element in every step. In every step values can be updated. After the finding min and max elements, to find max difference, we substract these values

* The time complexity of this algorithm in worst-case scenerio is O(n). This is because we should check the every element in the array in the loop which has a n-1 step. (which n is lenght of array A). So worst-case time complexity of this algoritm is O(n).