CS204: Design and Analysis of Algorithms

220001004, 220001005, 220001006

January 9, 2024

Example 1: Given, $f(n) = 10n^2 + 7n + 8$, prove that $f(n) = O(n^2)$.

Ans: We can see that,

$$f(n) \leq 10n^2 + 7n^2 + 8n^2 = 25n^2 \text{ (for } n \geq 1\text{)}$$

$\therefore f(n) = O(n^2)$ [$f(n) = O(g(n))$ if $0 \leq f(n) \leq c * g(n)$ for some $c,$ and $\forall n \geq n_o$.]
Here, $n_o = 1,$ and $c = 25$.
In general, $f(n) = an^2 + bn + c \leq (a + b + c)n^2 \ \forall \ n \geq 1$.
So, $f(n) = O(n^2)$ for $n_o = 1, k = (a + b + c)$.

Example 2: Disprove $n^3 \neq O(n^2)$.

Ans: Assume that the proposition is true. Then $n^3 \leq c \cdot n^2$ (for some $n_o,$ such that, it is valid $\forall n \geq n_o$).
But for $n \geq c + 1, (c + 1)^3 \geq c \cdot (c + 1)^2$. We arrive at a contradiction. So, the proposition is false, and $n^3 = O(n^2)$.

Example 3: If $f_1 = O(g_1(n))$, $f_2 = O(g_2(n))$. Prove that $f_1 + f_2 = O(max(g_1, g_2))$.
Ans:

$$f_1 \leq c_1 \cdot g_1(n) \text{ for } n \geq n_1$$

$$f_2 \leq c_2 \cdot g_2(n) \text{ for } n \geq n_2$$

consider $n_3 = max(n_1, n_2)$. Now, for all $n \geq n_3$,

$$f_1 + f_2 \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq (c_1 + c_2) \cdot max(g_1, g_2), \text{ as } g_1 \text{ and } g_2 \leq max(g_1, g_2).$$

Let $c_3 = c_1 + c_2$. So, $f_1 + f_2 \leq c_3 \cdot max(g_1, g_2)$ and therefore, $f_1 + f_2 = O(max(g_1, g_2))$ for $n \geq n_3$.

Example 4: Find time complexity. (Unique number detection)

```
1: for $i = 0$ to $n - 1$ do
2:     for $j = i + 1$ to $n - 1$ do
3:         if $A[i] = A[j]$ then
4:             return false
5:         end if
6:     end for
7: end for
8: return true
```

Ans: The total number of operations of the algorithm would be proportional to the number of times the loop runs.
For, $i = 0$ we have the inner loop (lines 2 to 6) run $(n - 1)$ times.
For, $i = 1$ we have $(n - 2)$ times.
For, $i = 2$ we have $(n - 3)$ times.

$\vdots$

For, $i = n - 2$ we have 1 times.
For, $i = n - 1$ we have 0 times.

Total time (worst case) is thus,

$$T(n) = 0 + 1 + 2 + \ldots (n-1) = \frac{n(n-1)}{2} = O(\frac{n^2}{2} - \frac{n}{2}) = O(n^2)$$

.

Example 5: Find time complexity. (Number of digits in a binary number)

```
count ← 1
while $n > 1$ do
    count ← count + 1
    $n \leftarrow n/2$
end while
```

Ans: For simplicity, assume $n = 2^i$. The value of n in the loop would progress as follows:

$$n, \frac{n}{2}, \frac{n}{2^2}, \ldots, \frac{n}{2^{i-1}}$$

This is because $\frac{n}{2^i} = 1$ and this will violate the while loop condition. The time complexity here would depend on the number of times the while loop runs which will be i and that is equal to $\log_2(n)$. Therefore, the time complexity becomes $O(\log(n))$.

Example 6: Find time complexity. (Insertion Sort)

```
for $i = 2$ to $A$.length do
    key ← $A[i]$
    $j \leftarrow i - 1$
    while $j \geq 1$ and key $> A[j]$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow$ key
end for
```

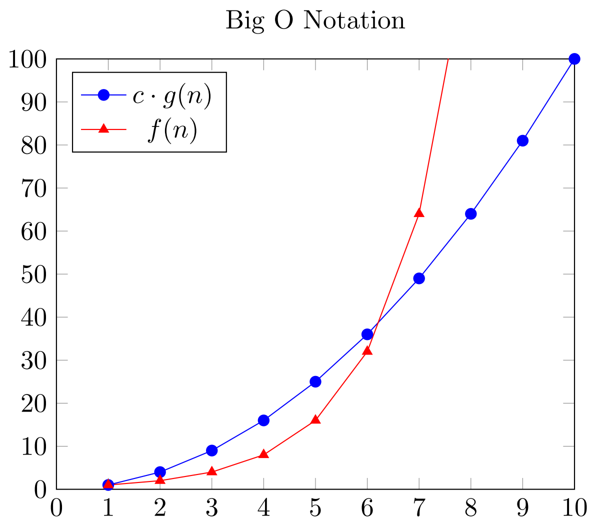
Ans: A line by line cost analysis is given:
1. $c_1 \cdot n$ (one extra for termination condition)
2. $c_2 \cdot (n-1)$
3. $c_3 \cdot (n-1)$
4. $\sum c_4 \cdot t_j$ (one extra for termination condition)
5. $\sum c_5 \cdot (t_j - 1)$
6. $\sum c_6 \cdot (t_j - 1)$
7. $c_7 \cdot (n-1)$

Here, c_i are some constants which denote the time cost of executing i^{th} statement. When multiplied by the number of times it is executed, we get the running time of that statement.
Also, t_j which denotes the number of times the inner WHILE loop runs and that depends on the input. The total time for running line 4 is $\sum c_4 \cdot t_j$, which has t_j not $(t_j - 1)$ because of the same reason for line 1, that is for a loop, the test is executed one time more than the loop body. The summation denotes (for line 4, 5, 6) that this is to be summed up for every iteration of the outer loop that therefore the

limits would be from $i = 2$ to $i = A.length$. Furthermore, this t_j would depend on the given input, that is if the array is sorted (descending here), then $t_j = 1$ as only the loop test needs to run and that will only run once. In the case when the array is in ascending order (worst case), comparisons will be done for the entire sorted subarray for which $t_j = j$. The algorithm's time complexity thus, is anything in between $O(n)$ to $O(n^2)$ depending on the input.

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + \sum c_4 \cdot t_j + \sum c_5 \cdot (t_j - 1) + \sum c_6 \cdot (t_j - 1) + c_7 \cdot (n-1)$$

2.6 Best-case Time Complexity



Definition: $\Omega(g(n)) = \{f(n) : \exists\, c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } 0 \le c \cdot g(n) \le f(n)\}$

- Worst-case is not a good measure as it is infrequent practically and average-case analysis is not always feasible (we do amortized analysis is such cases).
- Best-case complexity is helpful when the worst-case complexity is same, as that tells us about the average-case complexity.

2.7 Average-case Time Complexity

Definition: $\Theta(g(n)) = \{f(n) : \exists\, c_1 \in \mathbb{R}^+ \text{ and } \exists\, c_2 \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)\}$