Design and Analysis of Algorithms

Anurag (22)

Assignment 01

(Q1). What do you understand by Asymptotic notations. Define different Asymptotic notations with examples.

(A.) Asymptotic notations are set of mathematical tools used to describe the behaviour of function as their input sizes approach infinity. They are often used to analyse the time and space complexity of algorithms. There are mainly three types of asymptotic notations -

① Big O notation (O): This notation provides an upper bound on the growth rate of a function. It represents the worst case running time of an algorithm, which is the maximum amount of time it could take to complete. For example, we say that an algorithm has a time complexity of $O(n)$, we mean that the algorithm's running time grows at most linearly with the size of the input.

eg:-
```
int sum = 0;
for (int i=1; i<=n; i++) {
    sum = sum + i;
} // The time complexity is O(n)
```

② Omega notation (Ω): This notation provides a lower bound of the growth rate of a function. It represents the best-case running time of an algorithm, which is the minimum amount of time it could take to complete for example, if we say that an algorithm has a time complexity of $\Omega(n)$, we mean that the algorithm's running time grows at least linearly

with the size of its input.

③ Theta notation (Θ): This notation provides both an upper and a lower bound on the growth rate of a function, it represent the average case running time of an algorithm, which is the expected amount of time it would take to complete. For example: if we say that an algorithm has a time complexity of Θ(n), we mean that the ~~algorithm has a time complexity~~ algorithm's running time grows linearly with the size of its inputs, and these are no faster or slower growth rates

Eg:-
```
int bubbleSort(int a[], int size){
    for(int i=0; i<size; i++){
        for(int j=0; j<n-i-1; j++){
            if(a[j] > a[j+1])
                swap(a[j], a[j+1]);
        }
    }
    return a[];
```

⇒ the avg case time complexity is $\Theta(n^2)$.

(Q2). What should be the time complexity of

```
for (i=1; i<=n){ i = i*2}
```

(A.) The time complexity of the loop

```
for (i=1 to n) {
    i= i* 2;
}
```

can be determined by counting the number of iterations that the loop will execute as a function of the input size 'n'.

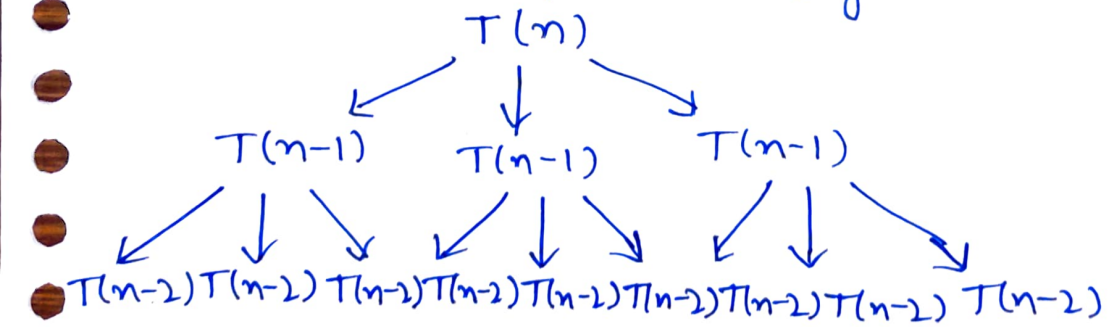
Here, the value of i is being denoted in each iteration, ~~top~~ loop terminates when i becomes greater than n.

$2^k = n$, $\therefore k <= \log(n)$

Time complexity = $O(\log n)$

(Q3). $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$

(A) The time complexity of Recursive function can be determined by analysing the number of function call it makes as a function of the input size 'n' Each call to T(n) results in 3 calls to T(n-1) until n reaches 0, at which point the function return 1. This can be represented using tree.



The height of tree is n, at each level there are 3 nodes.

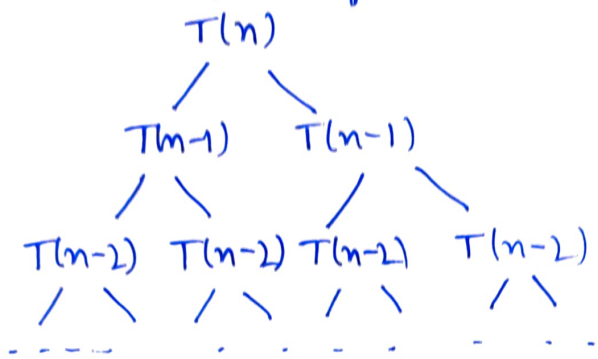
The total number of calls are 3^n.

Time complexity is $O(3^n)$.

Anurag (22)

(Q4). $T(n) = \{2T(n-1)-1$, if $n>0$, otherwise $1\}$

(A.) Input size n, Each T(n) result in 2 calls to T(n-1) until n reaches 0, at point ~~two~~ return 1



Height is n, at each level 2 nodes, Total function calls is 2^n. Time complexity $O(2^n) * O(1) = O(2^n)$.

(Q5). What should be the time complexity of

```
int i=1, s=1;
while (s<=n). {
        i++;
        s = s+i;
        printf ("#");
}
```

(A.) The time complexity of the given while loop is O(sqrt(n)). The loop iterates until the value of s becomes greater than n, At each iteration i is incremented by 1, and s is updated to s+i, the number of iterations required to reach $s>n$, $s+(i+1)>n$ is $i^2+i-2(n-s)>0$, this can be solve by quadratic formula.

(Q6). void function (int n) {
int i, count=0;
for (i=1; i * i <=n ; i++)
count++;
} }

(A.) The time complexity of the given function is O(sqrt(n)), the for loop iterates from i=1 to i*i <=n, the loop will execute for all values of ~~it~~ i from 1 to the largest integer less than or equal to the square root of n.

(Q7). void function (int n) {
int i,j,k, count=0;
for (i=n/2; i<=n; i++) {
for (j=1; j<=n; j=j*2) {
for (k=1; k<=n; k=k*2)
count++;
}
}
}

(A.) Time complexity is O(n² log(n)), the function consist of three nested loop that iterate over variable i, j and k the first loop take n/2 iteration, the second loop iterates over the variable j from 1 to n in powers of 2, which takes log2(n) iteration. The third loop iteration over the variable k from 1 to n in power of 2, which also takes log2(n) iteration.

(Q8). function (int n) {

```
    if (n==1)
        return;
    for (i=1 to n) {
        for (j=1 to n)
            printf("*");
    }
    function (n-3);
}
```

(A.) The function is a recursive function that is called with argument (n-3), it contain two nested loop that iterates over the variable it and j. The outer loop iterates n times, and the inner loop also iterates n time, n * n times at each recursive call, the value of n is decreased by 3. The function will be called a total of n/3 times recursive until n=1, time complexity is $O(n^2 (n/3)^2)$.

(Q9). void function (int n) {

```
    for (i=1 to n) {
        for (j=1; j<=n; j=j+1)
            printf ("*");
    }
}
```

(A.) The function is consist of two nested loop that iterate over the variable i and j the outer loop iterate over n times and inner loop iterate n/i times

$n + n/2 + n/3 + \cdots + 1$, this is Hermonic series

$\log(n) +$ $+ O(1/n)$

∴ The time complexity is $O(n \log(n))$.

(Q10). For the function, n^k and c^n, what is the asymptotic relationship b/w these function?
Assume that $k >= 1$ and $c > 1$ are constants. Find out the value of c and n_0 for which relation holds.

(A.) $n^k = O(c^n)$ as n approaches infinity.

n^k is bounded above by c^n.

Anurag (22)