Prakhar Pande
28
Section-D, University No: 2016906

Tutorial 2

1)

```
void func(int n)
{
    int j=1, i=0;
    while(i<n)
    {
        i+=j;
        j++;
    }}
```

Values after execution

1st time → i = 1
2nd time → i = 1+2
3rd time → i = 1+2+3
4th time → i = 1+2+3+4
for i^{th} time → $i = (1+2+3+4 \cdots i) < n$

$\Rightarrow i(i+1)/2 < n$
$\Rightarrow i^2 < n \Rightarrow i - \sqrt{n}$

∴ Time Complexity → $O(\sqrt{n})$ Ans

2) Recurrence Relation

$T(n) = F(n-1) + F(n-2)$

Let $T(n)$ denote the time complexity of $F(n)$.
For $F(n-1)$ and $F(n-2)$ time will be $T(n-1)$ and $T(n-2)$.
We've one more addition to sum as results. For $n > 1$

$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{---(1)}$$

For $n=0$ & $n=1$, no addition occurs
∴ $T(0) = T(1) = 0$

Let $T(n-1) \simeq T(n-2)$

Putting 2 in 1 we get.

$T(n) = T(n-1) + T(n-1) + 1$

$\Rightarrow 2T(n-1) + 1$

Using Backward Substitution :

$T(n-1) = 2T(n-2) + 1$

$T(n) = 2[2T(n-2)+1] + 1 = 4T(n-2) + 3$

We can substitute $T(n-2) = 2T(n-3) + 1$

$T(n) = 8T(n-3) + 7$

General equation :

$T(n) = 2^k T(n-k) + (2^k - 1)$ — (3)

For $T(0)$, $n-k=0 \Rightarrow k=n$

Substituting values in (3)

$T(n) = 2^n \times T(0) + 2^n - 1$

$\Rightarrow 2^n + 2^n - 1$

$\Rightarrow \boxed{T(n) = O(2^n)}$

Space complexity $\Rightarrow O(N)$

Reason :-

The function calls are executed sequentially. Sequential execution guarantees that the stack size will exceed the depth of caller. ~~From~~ For first $F(n-1)$ it will make N stack frames, the other $F(n-2)$ will create $N/2$ so the largest is N.

3) O(n log n)

```
#include <iostream>
using namespace std;

int partition(int arr[], int start, int end)
{
    int pivot = arr[start];
    int count = 0;
    for(int i = start; i <= end; i++)
        if(arr[i] <= pivot)
            count++;
    int pivot_ind = start + count;
    swap(arr[pivot_ind], arr[start]);
    int i = start, j = end;
    while(i < pivot_ind && j > pivot_ind)
    {
        while(arr[i] <= pivot) i++
    }
    while(arr[j] > pivot); --j;
    if(i < pivot_ind && j > pivot_ind)
        swap(arr[i++], arr[j--]);
    }
}
    return pivot_ind;
}
    void quick(int arr[], int start, int end){
    if(start >= end) return;
    int p = partition(arr, start, end)
    quick(arr, start, p-1);
    quick(arr, p+1, end);
}
```

```
int main()
{
  int arr[] = {6, 8, 5, 2, 1};
  int n = 5;
  quick(arr, 0, n-1);
  return 0;
}
```

> $O(N^3)$

```
int main()
{
  int n = 10
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      for(int k = 0; k < n; k++)
        printf("*");
  return 0;
}
```

> $O(\log(\log_2 n))$:-

```
int countPrimes(int n)
{
  if(n < 2) return 0;

  bool[] nonprime = new bool[n];
  nonprime[1] = true;
  int numNonPrimes = 1;
  for(int i = 2; i < n; i++)
    {
      if(nonPrime[i]) continue;
```

```
int j = i * 2;
while (j < n)
{
    if (!nonprime[j]) {
        nonprime[j] = true;
        numnonPrime++;
    }
    j += i; } }
return (n-1) - numNonPrime;
}
```

Ans 4 $T(n) = T(n/4) + T(n/2) + cn^2$

Akra Muster's Theorem
We can assume $T(n/2) \geq T(n/4)$
Equation can be rewritten as
$T(n) <= 2T(n/2) + cn^2$
$T(n) <= O(n^2)$
$T(n) = O(n^2)$
Also $T(n) \geq cn^2 \Rightarrow T(n) \geq O(n^2)$
$\Rightarrow T(n) = \Omega(n^2)$
$\therefore T(n) = O(n^2)$ & $T(n) = \Omega(n^2)$
$\boxed{T(n) = O(n^2)}$

Ans 5 For $i=1$, inner loop is executed n times.
For $i=2$, inner loop is executed n/2 times.
For $i=3$, inner loop is executed n/3 times.

It's forming a series :-
$n + n/2 + n/3 + \ldots + n/n$
$\Rightarrow n(1 + 1/2 + 1/3 + \ldots 1/n) \Rightarrow n \sum_{R=2}^{n} 1/R \Rightarrow n \log n$
$O(n \log n)$

6) for (int i = 2; i <= n; i = pow(i, k))
{
// loop O(n) expression or statements.
}

with n iterations :-
i take values
for 1st iteration → 2
for 2nd iteration → 2^k
for 3rd iteration → $(2^k)^3$
for n iterations → $2^{k(\log_k \log(n))}$

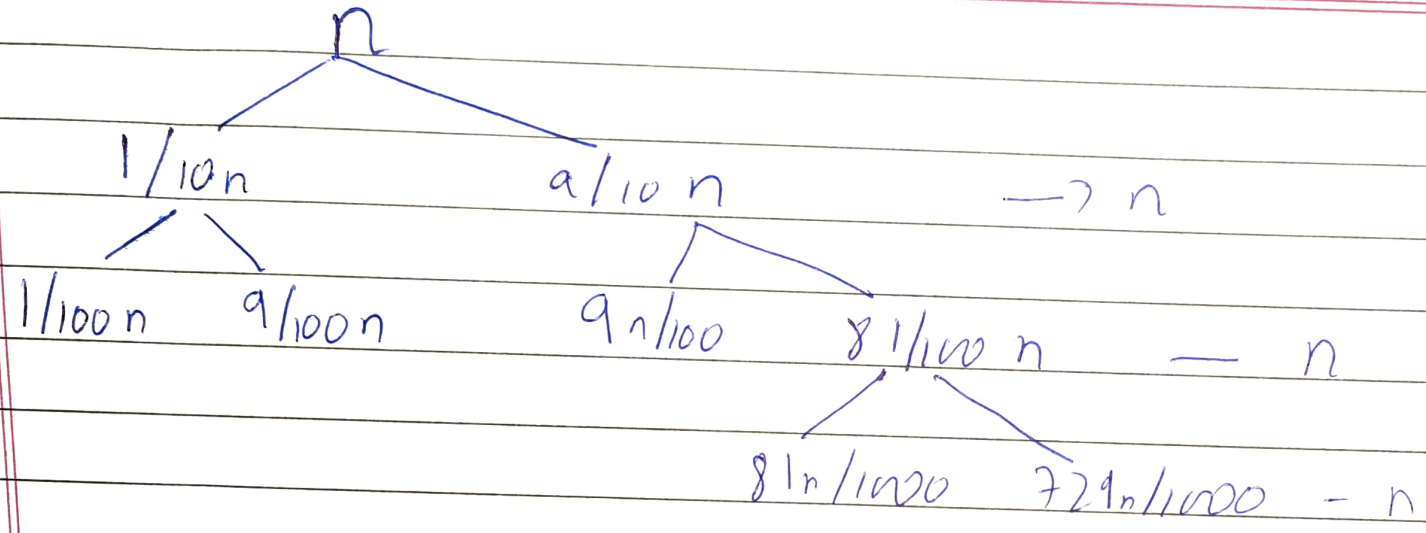
∴ last term must be less than equal as to n.

$$2^{k \log_k(\log(n))} = 2^{\log n} = n$$

Each iteration takes constant time.
∴ Total iteration = $\log_k(\log(n))$.

Time Complexity = $O(\log(\log(n)))$ Ans.

7)



7) we split in this manner.

Recurrence relation - $T(n) = T(9n/10) + T(n/10) + O(n)$

First branch is of size $9n/10$ & second one is $n/10$.

Solving the above using recursion tree approach calculating values

At 1st level, Value $= n$

At 2nd level, Value $= 9n/10 + n/10 = n$

Value remains same at all levels i.e. n

Time Complexity = Summation of all values

$= O(n \log_{10/9} n)$ Upper bound.

$= \Omega(n \log_{10} n)$ (lower bound)

$\Rightarrow O(n \log n)$ // Ans