Name : Jayesh Singh
Section : D
Roll Number : 11

Date

TUTORIAL - 2

1) What is time complexity of below code and how?

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

⇒

i	j	
0	1	(initial)
1	2	
3	3	~~1 + 3 + 6 + . . .~~
6	4	
⋮	⋮	
	n	

for i^{th} time ⇒ $i = (1+2+3+ \cdots i) < n$

$\Rightarrow \frac{i(i+1)}{2} < n$

$\Rightarrow i^2 < n$

$\Rightarrow i = \sqrt{n}$

Time complexity = $O(\sqrt{n})$

Date

2)
```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recurrence Relation

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

Let $T(n)$ denote time complexity of $F(n)$.

For $n \geq 1$

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

For $n=0$ & $n=1$, no addition occurs

$\therefore T(0) = T(1) = 0$

Let $T(n-1) \approx T(n-2)$ — (2)

(2) in (1)

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

Using backward substitution:

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

$T(n) = 2\times [2\times T(n-2) + 1] + 1$

$= 4T(n-2) + 3$

We can substitute

$T(n-2) = 2\times T(n-3) + 1$

$\Rightarrow T(n) = 8\times T(n-3) + 7$

General Equation:

$T(n) = 2^k \times T(n-k) + (2^k - 1)$ — (3)

for $T(0)$

$n-k=0 \Rightarrow k=n$

Date

substituting values in (3)

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

$$T(n) = O(2^n)$$

Space Complexity → O(N)

Reason :

Function calls are executed ~~scientif.~~ sequentially Sequential execution guarantees that stack size will never exceed the depth of calls.

3) O (n(logn))

// Merge Sort

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

void merge (int *array, int l, int m, int r)
{ int i, j, k, nl, nr;
  nl = m-l+1;  nr = r-m;
  int larr[nl], rarr[nr];
  for(i=0; i<nl; i++)
  { larr[i] = array[l+i]; }
  for(j=0; j<nr; j++)
     rarr[j] = array[m+1+j];
  i=0; j=0; k=l;
```

Date

```
while (i < nl  && j < nr)
{  if (larr[i] <= rarr[j])
   {  array[k] = larr[i];
      i++;  }
   else
   {  array[k] = rarr[j];
      j++;  }

      k++;
   }
}
void merge_sort(int *array, int l, int r)
{  int m;
   if (l < r)
   {  int m = l + (r-l)/2;
      merge_sort(array, l, m);
      merge-sort(array, m+1, r);
      merge(array, l, m, r);  }
}
```

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++) {
                  cout << "Hey";
         } } }
}
```

Date

O(log(logn))

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

  boolean [] nonprime = new boolean [n];
  nonprime [1] = true;
  int numnonprime = 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;
     }
}
```

Date

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

Using Master's Theorem,

$T(n) <= 2T(n/2) + cn^2$

$\Rightarrow T(n) <= O(n^2)$

$\Rightarrow T(n) = O(n^2)$.

Also, $T(n) >= (n^2)$ $\Rightarrow T(n) >= O(n^2)$

$T(n) = O(n^2)$

5) for $i=1 \rightarrow j = 1, 2, 3, \ldots n$

for $i=2 \rightarrow j = 1, 3, 5 \ldots$

for $i=3 \rightarrow j = 1, 4, 7 \ldots$

$T(n) = n + n/2 + n/3 + \ldots$

$= n(1 + 1/2 + 1/3 + \ldots)$

$\Rightarrow O(n\log n)$

Date

6) for first iteration $i = 2$
2nd iteration $i = 2^k$
3rd iteration $i = (2^k)^k = 2^{k^2}$

⋮

nth iteration $i = 2^{k^i} = n$
using logarithm,

$i = \log_k(\log n)$