Jaidev Rawat Sec - DS - 34 2015170

Q1 ⇒ Asymptotic means towards infinity, so in general there are notations that are used to tell the complexity of an algorithm. when the input is very large.

1) Big-oh (O) → $f(n) = O(g(n))$
   $g(n)$ is "tight" upper bound of $f(n)$

2) Big-Omega (Ω) → $f(n) = \Omega(g(n))$
   here $g(n)$ is tight lower bond of $f(n)$

3) Theta (θ) = $f(n) = \theta(g(n))$
   iff
   $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

Theta gives the tight upper and lower bond both.

Q2 ⇒ $i = i * 2$.

1 2 4 8 16 ... n

$a = 1$; $r = \frac{t_2}{t_1} = 2$

let it be k terms

$t_k = ar^{k-1} \Rightarrow n = 1 * 2^{k-1}$

$k = \log_2(2n) \Rightarrow \log_2(n)$ //

(3) ⇒ $T_1(n) = \{3T(n-1)\}$

$T(0) = 1$

$T(n-1) = \{3T(n-1-1)\}$

$T_2(n) = \{3^2 T(n-2)\}$

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

$T_3(n) = \{3^3 T(n-3)$

for kth term

$T_k(n) = \{3^k T(n-k)\}$

As $T(0) = 1$

$n-k = 0$; $n = k$

$= 3^n (T(0))$

$= 3^n$ //

(4) ⇒ $T_2(n) = \{2T(n-1)-1$ if $n>0$

$T(0) = 1$

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

$T_2(n) = 2[2T(n-2)-1]-1$

$= 4T(n-2)-2-1$

$T(n-2) = 2T(n-3)-2.$

$T_3(n) = 8T(n-3) - 4 - 2 - 1$

$$T_k(n) = 2^k[T(n-k)) - 2^{k-1} - 2^{k-2} \cdots 2^0$$

when $(n-k) = 0$; $n = k$

Putting $n = k$.

$= 2^n - 2^{n-1} - 2^{n-2} \cdots - 1$

$= 2^n - [2^0 + 2^1 + 2^2 + \cdots 2^{n-1}]$

$= 2^n - \frac{[1[2^n - 1]}{2-1}$ $\therefore a\left[\frac{r^n - 1}{r-1}\right]$

$= -1$

$T(n) = 1$

(5) $\Rightarrow$ $S \to 1$ to n

i++ $S = S + i$.

$S \to 1 \quad 3 \quad 6 \quad 10 \quad 15 \quad 21 \cdots n$ $i = 2$.

$i \to +2 \quad +3 \quad +4 \quad +5 \quad +6$ $i = 3$ $i = 4$

$S_0 \to S_i = S_{i-1} + i$

So at kth iteration. if while loop terminates

then $1 + 2 + 3 \cdots + k = \frac{k(k+1)}{2} > n$

$k = O(\sqrt{n})$.

(6) ⇒

1 to n i*i <= n ; i++

i ⇒ 1 2 3 4 5 ——

i*i ⇒ 1 4 9 16 25 -- n.

So at kth term if

$k^2 > n$ loop terminates

$k = \sqrt{n}$

7 ⇒ for(k=1 ; k<=n; k=k*2)

k = 2^0 2^1 2^2 2^3 -- 2^f

last term of GP if = ar^{n-1}

$= 1 \cdot 2^{(f-1)}$

$= 2^{f-1}$

So $2^{f-1} = n \Rightarrow f = \log_2 n$

for(1 to n j=j*2 ⇒ $\log_2 n$

for(i=n/2; i<=n; i++)

n/2 to n ⇒ $n - \frac{n}{2} + 1$ terms

$= \frac{2n - n + 2}{2} = \left(\frac{n+2}{2}\right)$

⇒ $(n)(\log^2 n)$

(9) ⇒

(15) → 1 → 1 to n → n steps

2 → 1 to n → $\frac{(n+1)}{2}$ steps.

3 → 1 to n $\frac{n+1}{3}$

4 → 1 to n $\frac{n+1}{4}$

Using approximation

$n \times \left[1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots \right]$ ($\frac{1}{n}$ terms)

log n

k terms = n terms

$n \log n$.

---

(8) → n^2 for nested loop

and for recursive function $(n-3)$

≈ function (n) thus n^3.

(11) ⇒

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

i	0	1	3	6	10	...	n.
j	1	2	3	4	5		

At kth index Sum is $0+1+2+3+\cdots k$

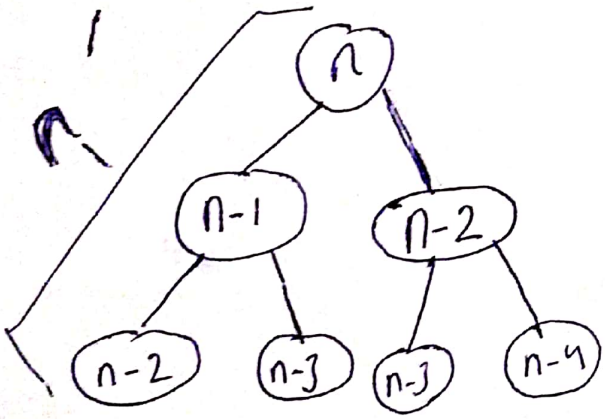
$\Rightarrow \frac{k(k+1)}{2} = n$

$k = \sqrt{n}$.

12 ⇒

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

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



1 + 2 + 4 + ... + 2^n

$$\Rightarrow \frac{1(2^{n+1} - 1)}{2 - 1} = [2^{n+1} - 1]$$

$$\Rightarrow O(2^n)$$

Space complexity will be $O(n)$ because the space required is proportional to maximum depth of the recursion tree.

(13) $n \log n$.

```
int sum = 0;
for (int i = 0 ; i < n ; i++)
{
    for (int j = 0 ; j < n ; j = j * 2)
    {
        sum += i;
    }
}
```

$n \wedge 3$

```
for (int i = 0; i < n ; i++)
{
    for (int j = 0; j < n ; j++)
    {
        for (int k = 0; k < n ; k++)
        {
            sum = sum + 1;
        }
    }
}
```

$\log(\log n)$

```
for (int i = 2; i <= n ; i = pow(i, k))
{
    ++i;
}
```

$2^{n/2}$

(14) →

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

$T(1) = 1$. As we can assume that

$T(n/4) <= T(n/2)$

So eq can be rewritten as

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

On applying masters method we get

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

(16) ⇒

```
for (int i = 2; i <= n; i = pow(i, k))
{
}
```

$i \Rightarrow 2, 2^k, (2^k)^k \Rightarrow 2^{k^2}, 2^{k^3} \dots 2^{k \log_k(\log(n))}$.

The last term must be less than or equal to n, and we have $2^{k \log_k(\log(n))} = 2^{\log(n)} = n$ which correctly agrees with the value of our last term. So total $\log_k(\log(n))$ operations.

(19) =)

```
int key = 100;  int flag = -1;
for(int i=0; i< n; i++)
{
    if (arr[i] == key)
    {
        flag = i
        break;
    }
}
if (flag == -1)
    cout << "not found";
else
    cout << "index = " << flag;
```

20 ⇒ Iterative Insertion Sort.

```
for (int i=1; i<n; i++)
{
    int value = arr[i];
    int j = i;
    while (j>0 && arr[j-1] > value)
    {
        arr[j] = arr[j-1];
        j--;
    }
    arr[j] = value.
}
```

Recursive insertion Sort

```
void insertion_sort (int arr[], int i, int n)
{
    int value = arr[i]; int j = i;
    while (j>0 && arr[j-1] > value)
    {
        arr[j] = arr[j-1];
        j--;
    }
    arr[j] = value;
    if (i+1 <= n)
        insertionSort (arr, i+1, n);
}
```

Insertion Sort is an online algorithm as it processes its input piece by piece in a serial fashion. which means if a new element is introduced in the array then it will placed in its right place. even if the sorting already had started.

Rest other Bubble Sort, Selection Sort, merge Sort, Quick Sort are offline Sorting algo.

21 ⇒ Sorting time complexities :-

Bubble Sort ⇒ n^2

Selection Sort ⇒ n^2

Insertion Sort ⇒ Best case ⇒ n
Average case ⇒ n^2

Quick Sort ⇒ Best and Average case ⇒ $n \log n$
Worst case ⇒ n^2

Merge Sort ⇒ $n \log n$.

22 ⇒

	Inplace	Stable	Online.
Bubble	Inplace	Stable	Offline
Selection	Inplace	Not Stable	offline
Insertion	Inplace	Stable	online
Quick	Inplace	Not Stable	offline
Merge	Not Inplace	Stable	offline.

(23) ⇒ Iterative binary search.

```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m <- (l + r) / 2;
        if (arr[m] = x)
            return m;
        if (arr[m] < x)
            l <- m + 1;
        else
            r <- m - 1;
    }
    return -1;
}
```

Time → $\log_2 n$

Space → $O(1)$

Recursive Binary Search

Time → $\log_2 n$

Space → $\log_2 n$

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid <- (l + r) / 2;
        if (arr[mid] = x)  return mid;
        else if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);
        else return binarySearch(arr, mid+1, r, x);
    }
    return -1;
}
```

24 → Recurrance relation for binary recursive search :-

$$T(n) = T\left(\frac{n}{2}\right) + 1.$$

which means at every step problem is divided into two halves.