Name - Muskaan Chaturvedi
Section - D3
University Roll - 2015177

Page: 1
Date: / /

Ans 1 Asymptotic notations are a mathematical way of representing time complexity

☆ They are notations used to represent complexity of an algorithm. In time analysis, instead of executing the algorithm, we have to analyse it.

☆ To calculate time without execution, number of times instructions are executed is considered and represented through Asymptotic notation.

☆ Important to properly represent time complexity of each algorithm and also to be able to compare which algorithm is better and find best one

→ Big-O (O)
$f(n) = O(g(n))$
means $g(n)$ is 'tight' upper bound of $f(n)$. $g(n)$ giving upperbound of $f(n)$. $f(n)$ can never go beyond $g(n)$
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$
$\forall n > n_0$, some constant $c > 0$.

→ Big omega (Ω)
$f(n) = \Omega(g(n))$ iff $f(n) > c \cdot g(n)$
$\forall n > n_0$ and some constant $c > 0$.
Means $g(n)$ is 'tight' lowerbound of $f(n)$. $f(n)$ will never perform better than $g(n)$.

→ Theta (Θ) –
$f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2(g(n))$. $\forall n > n_0$ where (n_1, n_2)
Same constant c_1 & $c_2 > 0$.
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
Only then upper and lower bound only a bound in which function will function. $f(n)$ can come go beyond $c_2(g(n))$ and can never go below $c_1(g(n))$.

→ small oh (o)
$f(n) = o(g(n))$
means $f(n) < c\, g(n)$ for all $n > n_0$ and for some constant $c > 0$.
Only upper bound of $f(n)$.

→ small omega (ω)
$f(n) = \omega(g(n))$
means $f(n) > c\, g(n)$ $\forall n > n_0$ and for some constant $c > 0$.
$f(n)$ can never remain between or equal to $g(n)$ after threshold no.

Ans2 for ($i = 1$ to n)
& $i = i * 2$ }
$i = 1, 2, 4, 8, 16, \ldots n$. This is a GP.
Assuming n is power of 2 as all are in power of 2.
$2^0, 2^1, 2^2, 2^3, 2^4, \ldots 2^k$.
Here first term of GP i.e $a = 1$.
Common ratio $= \frac{t_2}{t_1} = \frac{2}{1} = 2$ //.

To obtain kth term in GP, formula -

$$t_k = a r^{k-1}$$

kth term is n.

So $n = 1 \times 2^{k-1} = \frac{2^k}{2}$.

$$2n = 2^k$$

Taking Log on both sides

$$\log_2 n + \log_2 2 = k \log_2 2$$

[$\because \log(a \times b) = \log a + \log b$]

[$\because \log_a a = 1$ so $\log_2 2 = 1$]

$$k = \log_2 n + 1$$

Time Complexity = $O(\log_2 n + 1)$

Equivalent lower order term

Time Complexity = $O(\log_2 n)$

Ans 3 $T(n) = 3T(n-1)$ if $n > 0$, otherwise 1

$T(n) = 3T(n-1)$ ——— (1)

Using Backward substitution

Putting $n = n-1$ in (1)

$T(n-1) = 3T(n-1-1)$

$T(n-1) = 3T(n-2)$ ——— (2)

Putting value of $T(n-1)$ in (1)

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

$T(n) = 3^2(T(n-2))$ ——— (3)

Putting $n = n-2$ in (1)

$T(n-2) = 3T(n-2-1)$

$T(n-2) = 3T(n-3)$ ——— (4)

Putting value of $T(n-2)$ in (3),

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

$T(n) = 3^3 T(n-3)$ ——— (5)

Generalising it
$T(n) = 3^n T(n-n)$
$T(n) = 3^n T(0)$
$T(n) = 3^n$ $(\because T(0) = 1)$
Time complexity $= O(3^n)$.

Ans4 $T(n) = 2T(n-1) - 1$ if $n > 0$ else 1.
Using backward substitution
$T(n) = 2T(n-1) - 1$. ——— (1)
Putting $n = n-1$ in (1).
$T(n-1) = 2T(n-1-1) - 1$
$T(n-1) = 2T(n-2) - 1$ ——— (2)
putting value of $T(n-1)$ in (1)
$T(n) = 2(2T(n-2) - 1) - 1$
$T(n) = 2^2 T(n-2) - 2 - 1$ ——— (3).
putting $n = n-2$ in (1),
$T(n-2) = 2T(n-2-1) - 1$
$T(n-2) = 2T(n-3) - 1$ ——— (4)
putting value of $T(n-2)$ in (3),
$T(n) = 2^2(2T(n-3) - 1) - 2 - 1$
$T(n) = 2^3 T(n-3) - 4 - 2 - 1$ ——— (5).
Generalising it,
$T(n) = 2^n(T(n-n)) - 2^{n-1} - 2^{n-2} \ldots - 2^0$.
$= 2^n - 2^{n-1} - 2^{n-2} \ldots - 2^0$ [$\because T(0) = 1$]
$= 2^n - (2^n - 1) = 1$.
Time complexity $= O(1)$.

Ans5
```
int i = 1, s = 1;
while (s <= n){
    i++; s = s + i;
    printf("#");
}
```

Value of s would be
$S = 1, 3, 6, 10, \ldots n$.

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

$$n = \frac{K^2 + K}{2}$$

Ignoring lower order terms,

$$K^2 = n$$

$$K = \sqrt{n}$$

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

Ans 7.

```
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++;
}
```

j loop executes $\log n$ times and k loop executes $\log n$ times. i loop runs $n/2$ times.

Time complexity $= O\left(\frac{n}{2} \times \log n \times \log n\right)$

$= O\left(\frac{n}{2} \log^2 n\right) = O(n \log^2 n)$.

Ans 8.

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

i increments 4 times, j increments n times and function (n-3) means n/3 times running also.

Time complexity = $O\left(\frac{n}{3} \times n \times n\right)$.

$= O\left(\frac{n^3}{3}\right) = O(n^3)$.

Ans9

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

i → j

1 → n times
2 → n/2 times
3 → n/3 times
⋮
n → n/n times

So Time complexity = O(nlogn).

Ans10 As polynomials grow slower than exponentials, n^k has an asymptotic upper bound of $O(a^n)$.

Ans11

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

i	j
1	2
3	3
6	4

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

$k^2 = n \Rightarrow k = \sqrt{n}$.

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

Ans 12

For fibonacci series,

$T(0) = 0$

$T(1) = 0$

Recursive relation is

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

Assuming $T(n-1) \simeq T(n-2)$ so

$T(n) = 2T(n-1) + 1$ ——— (1)

Using Backward substitution

Put $n = n-1$ in (1)

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

$T(n-1) = 2T(n-2) + 1$ ——— (2)

Put value of $T(n-1)$ in (1)

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

$T(n) = 2^2 T(n-2) + 2 + 1$ ——— (3)

Put $n = n-2$ in (1)

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

$T(n-2) = 2T(n-3) + 1$ ——— (4)

Put value of $T(n-2)$ in (3)

$T(n) = 2^2(2T(n-3)+1) + 2 + 1$

$T(n) = 2^3 T(n-3) + 4 + 2 + 1$ ——— (5)

Generalising it,

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

We know that $T(0) = 0$

$n - k = 0$ so $n = k$.

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

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

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

Time complexity $= O(2^n)$.

Ans 13 (a) complexity = n

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

(b) complexity = n^3.

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

(c) complexity = log(log(n)).

```
void function (int n)
{
    int i, k;
    for (i=n; i>1; i=pow(i,k))
    {
        printf("Hello");
    }
}
```

Ans 14

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + cn^2.$$

Assuming $T\left(\frac{n}{2}\right) >= T\left(\frac{n}{4}\right)$.

$T(n) = 2T(\frac{n}{2}) + cn^2$

Using master's method

$c = \log_b a = \log_2 2 = 1$

$\Rightarrow n^c < f(n)$.

Time complexity $= O(n^2)$

Ans 15

```
int function(int n){
  for(i=1; i<=n; i++){
    for(j=1; j<=n; j+=i){
      //some O(1) task
    }
  }
}
```

i	j
1 →	n times
2 →	n/2 times
3 →	n/3 times
⋮	
n →	n/n times

i runs n times.

j runs log n times

Time complexity $= O(n \log n)$

Ans 16

```
for(i = 2; i<=n; i=pow(i,k))
{
  Some O(1) task
}
```

k is constant here. So values of i may be

$i = 2, 2^k, (2^k)^k, (2^{k^2})^k \ldots 2^{k^{\log_k(\log_2 n)}}$

So $2^{k^{\log_k(\log_2 n)}} = n$.

$2^{\log(1)} = 1$.

Time complexity = $O(\log(\log n))$

Ans 17 Recurrence Relation —

$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + O(n)$

Taking one branch 99% and other 1%. So

$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right) + O(n)$

first level = n

second level = $\frac{99n}{100} + \frac{n}{100} = n$

Hence this level remains same for any kind of partition.

If longest branch take k then $O(n \log \frac{100}{99n})$ and for a smallest branch $\Omega(n \log_{10} n)$.

Either way the base complexity of $O(n \log n)$ remains same

Ans 18 (a) n, $n!$, $\log n$, $\log \log n$, $\log n^2$, $\log n!$, $n \log n$, $\log 2^n$, 2^n, 2^{2^n}, 4^n, n^2, 100.

→ $100 < \sqrt{n} < \log(\log n) < \log n < 4^{\log n} < n \log n < \log n! < n^2 < n! < 2^n < 4^n < 2^{2^n}$.

(b) $1 < \log\log n < \sqrt{\log n} < \log n < \log 2n < n < n\log n < 2n < 2(2^n) < n! < n2$

Q $6 < \log_2 n < \log n! < n\log_2 n < n\log_6 n^2 < 5n \leq n! < 8n^2 < 7n^3 < 2^{2n}$

Ans 19
```
linearsearch(arr, n, key)
{
    for(i=0 to n-1)
    {
        if(arr[i]==key)
            return i;
            break;
    }
    return -1;
}
```

Ans 20 Iterative insertion sort

```
insertion(int arr, int n)
{
    for(i=1; i<n; i++)
    {
        int temp = arr[i];
        int j = i;
        while(j>0 && arr[j-1]>val)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = temp;
    }
}
```

Recursive Insertion Sort

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

☆ Insertion sort is an example of online sorting where as elements are coming, then are able to arrange array in sorted order. When element comes in insertion sort, just put it at right position after comparing.

Ans 21

Sorting algo	Best case	Average case	Worst case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

Ans22 Bubble sort, insertion sort, merge sort are stable algorithms

☆ Bubble sort, selection sort, insertion sort and Heap sort are inplace sorting algorithms

☆ Insertion sort is an online sorting algorithm.

Ans23
```
int binarysearch(int arr[], int l, int r,
                 int key)
{
    while (l <= r) {
        int mid = (l + r) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            l = mid + 1;
        else
            r = mid - 1; }
    return -1; }.
```

Time complexity of iterative Binary search is - Best case = O(1) when mid element is key to be found. Average and worst case = O($\log_2 n$). No extra array or memory used in iterative method so space complexity = O(1)

Ans24 Recurrence relation for recursive binary search is -

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