(1) Asymptotic notations ~~are~~ is the evaluations of performances of an algorithm in terms of input size not actually running time, we calculate how the time taken by algorithm increases with input size.

Types:

=) Big O notation: It represents the upper bound of the running time of an algorithm i.e, worst case.

=) $f(n) = O(g(n))$

means that growth rate of $f(n)$ is asymtotically less than or equal to growth of $g(n)$.

=) Omega Notation (Ω): represents the lower bound of running time of an algorithm i.e, best case complexity.

$\Rightarrow f(n) = \Omega(g(n))$

Means growth of $f(n)$ is greater than or equal to $g(n)$'s growth.

2) Theta Notation (Θ-notation), it encloses the function from above and below as it shows both upper & lower bounds of running time. i.e, used in average case.

$$f(n) = \Theta(g(n))$$

Means growth rate of $f(n)$ = growth rate $g(n)$.

(2) $i = 1, 2, 4, 8, 16 \ldots n$

as it increases in logarithmic terms of base 2,

$\Rightarrow O(\log_2 n)$.

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

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

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

$$\ldots$$

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

$$\ldots$$

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

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

$$= 3^n \text{ or } O(3^n)$$

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

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

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

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

$= 2^3 (T(n-3)) - 2^2 - 2^1 - 2^0$

$= 2^n \, \& \, T(n-n) - 2^{n-1} - 2^{n-2} \ldots - 2^0$

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

$= 1$ & OR $O(1)$

(5) $i = 1$, $s = 1$, $i = 2$, $s = 3$, $i = 3$, $s = 6$, $i = 4$, $s = 10$, $i = 5$, $s = 15$.

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

(6) $i = 1$, $i = 4$, $i = 9$, $i = 16$, $i = 25$

$\therefore O(\sqrt{n})$

(7) loop for i * & loop of j * loop for k

$O(\cancel{n^2}\; n * \log n * \log n)$

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

(8)(9) The code represents:

$T(n) = \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \ldots + \frac{n}{n}$

$n(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots + \frac{1}{n})$

$= \Theta\; n(\log n)$

$\therefore O(n \log n)$

(11) $i = 1, 3, 6, 10, 15 \ldots n$

$\therefore O(\sqrt{n})$ Ans. as $\left[\frac{k(k+1)}{2}\right] =$

$1 + 2 + 3 + \ldots k$

where k = total no. of iterations.

(14) Using Master's theorem,

Assuming $T(n/2) = T(n/4)$,

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

Master's theorem on right side.

$T(n) <= \Theta(n^2)$ OR $T(n) = O(n^2)$, also.

$T(n) >= cn^2$ i.e, $T(n) = \Omega(n^2)$

$\therefore T(n) = O(n^2)$ & $T(n) = \Omega(n^2)$

$T(n) = \Theta(n^2)$

(13) ~~$\Theta(n \log(n))$~~ $O(\log \log n)$

```
#include <iostream>
using namespace std;
void main()
{
    for (int i=2; i<=n; i*=i)
        cout << i << " ";

    cout << endl;

}
```

=) O(n log n)

```
#include <iostream>
using namespace std;
void main()
{
    for(int i=1 ; i<=n; i++)
    {
        for(int j=1 ; j<=n; j*=2)
        {       cout << j << " ";

        cout << endl;

    }
}
```

=) O (n^3)

```
#include <iostream>
using namespace std)
void main()
{   int c=1;
    for(int i=1; i<=n; i++)
    {
        for(int j=1; j<=n; j++)
        {
            for(int k=1; k<=n; k++){
                cout << c << " ";
                c++;
            }
        }
    }
}
```

(15) The ~~se~~ question generates harmonic series $\times n$.

i.e,

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \cdots + \frac{n}{n}$$

$$\Rightarrow n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots + \frac{1}{n}\right)$$

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

(16) $i = 2,\ 2^{k},\ (2^{k})^{k},\ 2^{k^3},\ 2^{k^4} \ldots$

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

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

$\therefore\ T = O(\log \log n)$

(18)

a) log log n, log n, log(n!), root(n), 100,
n, n log n, n^2, 2^n, 2^{2n}, 4^n, n!

(19)

```
for( i to len of array)
{
    if( a[len] < num  OR  arr[i] >= num)
    {
        if( arr[i] == equal to num)
            print " Number present
        else
            print " Number not present"
        break;
    }
}
```

(20)

```
void isort (int a[], int n)
{
    if (n <= 1)
        return;
    isort (a, n-1);
    int v = a[n-1];
    int p = n-2
    while (p >= 0 && a[p] > v)
    {
        a[p+1] = a[p];
        p = p-1;
    }
    a[p+1] = v;
}
```

It considers only one input per iteration and produces partial solution without considering future results hence called online algo.

(21)

Algo	Best time	Avg time	Worst time	Worst space.
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort.	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$.

(22)

	Implace	Stable	Online
Bubble sort	✓	✓	✗
Selection sort	✓	✗	✗
Insertion sort	✓	✓	✓
Quick sort	✓	✗	✗
Merge sort	✗	✓	✗
Heap sort	✓	✗	✗

(23) Recursive binary Search.

```
int binary (int a[], int low, int high, int s)
{
    if (low > high)
        return -1;
    int mid = (low+high)/2;
    if (~~…~~ s == a[mid]
        return mid;
    else if( s < a[mid]
        return binary(a, low, mid-1, s);
    else
        return binary(a, mid+1, high, s);
```

	Time	Space
Linear Search	$O(n)$	$O(1)$
Binary Iterative Search	$O(\log n)$	$O(1)$
Binary Recursive Search	$O(\log n)$	$O(\log n)$

(24) $T(n) = T(\frac{n}{2}) + 1$ — (1), following manner.

$n = n/2$,

$T(\frac{n}{2}) = T(\frac{n}{4}) + 1$ — (2)

Putting (2) in (1)

$T(n) = T(\frac{n}{4}) + 2$

now $\frac{n}{4}$ in (1).

$T(\frac{n}{4}) = T(\frac{n}{8}) + 1$ — (3)

(3) in (1).

$T(n) = T(\frac{n}{8}) + 3$

$T(n) = T(\frac{n}{2^k}) + k$ — (4)

$\frac{n}{2^k} = 1$

$2^k = n$, taking log,

$k \log_2 2 = \log_2 n$

$k = \log_2 n$, k in (4).

$$T(n) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n$$

$$T(n) = T\left(\frac{n}{n}\right) + \log_2 n$$

$$T(n) = T(1) + \log_2 n$$

$T(1) = 1,$

$\therefore \quad T(n) = 1 + \log_2 n$

$\Rightarrow O(\log_2 n)$ Ans