Name : Abha Garg .
Section : A
Subject : Design and Analysis of Algorithms.
Roll no : 2014508 .

Q1. Asymptotic notations are the mathematical notations used to describe running time of an algorithm when the input tends towards a particular limiting value. These notations are generally used to determine the running time of an algorithm and it grows with the amount of input.

There are five types -

i) <u>Big Oh(O) Notation</u> : Big oh notation defines an upper bound of an algorithm. It bounds the function only from above.

formally : $O(g(n)) = \{ f(n)$: there exits positive constant c ~~and~~ n_0 such that $0 \leq f(n) \leq g(n) \ \forall \ n \geq n_0 \}$.

ii) <u>Small Oh(o) Notation</u> - We denote o- notation to denote an upper bound that is not asymptotically tight.

Formally, $o(g(n)) = \{ f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c\,g(n) \ \forall \ n \geq 0 \}$

3) <u>Big Omega (Ω) Notation</u> - The big omega denote asymptotic lower bond more formally:

$\Omega(g(n)) = \{ f(n)$: for any position constant c & n_0 $0 \leq c\,g(n) \leq f(n) \ \forall \ n \geq n_0 \}$.

4) Small Omega (ω) Notation : By analogy, ω notation is to Ω notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. Formally:

$$\omega(g(n)) = \{ f(n) : \text{for any positive } c > 0, \text{ there exists } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \ \forall\ n \geq n_0 \}$$

5) Theta (Θ) Notation - The theta notation bounds the function from above and below. So it defines exact asymptotic behaviour.

Formally: $\Theta(g(n)) = \{ f(n) :$ there exists positive constants c_1, c_2 & n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall\ n > n_1 \}$

2) $\Theta(\log n)$

3) $T(n) = \begin{cases} 3T(n-1) & \text{if } n > 0 \\ 1 & n \leq 0. \end{cases}$

By using back substitution.

$T(n) = 3T(n-1)$ — ①

$T(n-1) = 3T((n-1)-1) = 3T(n-2)$ — ②

$T(n-2) = 3T(n-2-1) = 3T(n-3)$ — ③

Putting eqn ③ in ②.

$T(n-1) = 3(3T(n-3))$ — ④

Putting ④ in ①.

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

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

Let $k = n$

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

$\approx O(3^n)$

4) $T(n) = \begin{cases} 2T(n-1)-1 & n > 0. \\ 1 & n \leq 0. \end{cases}$

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

using back substitution

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

Putting (2) in (1).

$T(n) = 2(2T(n-2)-1)-1$ — (3)

From eqn (1)

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

Putting eqn (4) in (3).

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

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

$2(4T(n-3)-2-1)-1$

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

$T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} \ldots -1$

Let $k = n$.

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

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

$= 2^n - [1+2+4+ \ldots 2^{n-1}]$

$= 2^n - 2^{n-1} - 1 \approx O(2^n)$

5)
```
i = 1, s = 1.
while (s <= n)
{ i++;
  s = s + i;
  printf("#");
}
```

O(n);

6)
```
void func(int n)
{ int i, count = 0;
  for (i = 1; i * i <= n; i++)
      count++;
}
```

$O(\sqrt{n})$

7)
```
void func(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 ++;
}
```

O(n logn logn)

8. The recurrence relation is

$$T(n) = \begin{cases} T(n-3) + n^2 & n > 1 \\ 0 & n <= 1 \end{cases}$$

solving by back subsitution

$T(n) = T(n-3) + n^2$ — (1)

$T(n-3) = T(n-6) + (n-3)^2$ — (2)

$T(n-6) = T(n-9) + (n-6)^2$ — (3)

Substituting (3) in (2) and (2) in (1).

$T(n) = T(n-9) + (n-6)^2 + (n-3)^2 + n^2$

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

Let $3k = n$

$k = n/3$.

$T(n) = T\left(n - 3 \times \frac{n}{3}\right) + \underbrace{n^2 + (n-3)^2 + \dots + \left(n - 3\left(\frac{n}{3} - 1\right)\right)^2}_{k \text{ terms}}$

$T(n) = T(1) + n^2 + (n-3)^2 + (n-6)^2 + \dots + (n-n+3)^2$

Taking only higher order terms, n^2 will be obtained k times.

$\therefore$ n^2 will be obtained $n/3$ times.

$T(n) = 1 + (n^2 + n^2 + n^2 + \dots) + (xn + yn + zn \dots)$

$T(n) = 1 + kn^2 + \dots$

$T(n) = 1 + \frac{n^3}{3}$

$\approx O(n^3)$

Q 9

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

$$n + \frac{n}{2} + \frac{n}{3} + \ldots \frac{n}{n}$$

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

This ~~same~~ sum will converge to $\log n$.

Hence $\Theta(n \log n)$

10. $f(n) = n^k$ $\quad k >= 1$

$g(n) = a^n$ $\quad a > 1$

Exponential functions grow faster than polynomial functions here.

$$O(n^k) < O(a^n)$$

for values of $k \geq x$ and $a \geq y$. Let's calculate x & y.

Let $k = 2$ & $a = 2$ as well.

$$f(n) = n^2, \; g(n) = 2^n$$

Take log on both sides.

$\log(n^2) = 2\log_2(n)$ $\qquad$ $\log(g(n)) = n \log_2 2$

$$O(\log n) < O(n)$$

Hence for $k >= 2$ and $a >= 2$ the condition satisfies.

(11) $O(\sqrt{n})$, because the value of i goes as follows:

1, 3, 6, 10, 15, 21 . . .

Also we know that

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

for the sum of series $1+2+3+4 \ldots$

So the series 1, 3, 6, 10, 15 will stop where a_n becomes equal to or greater than n.

$$\therefore \frac{n(n+1)}{2} = n_0 \leftarrow \text{final value of } n$$

$$n \approx \sqrt{n_0}$$

12)

$$T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & n \geq 2 \\ 1 & 0 \leq n < 2 \end{cases}$$

Assume time taken by $T(n-2) \simeq T(n-1)$

So $T(n) = 2T(n-2) + C$

Solving we get:

$T(n) = 2 \cdot 2 \cdot 2\, T(n - 2 \cdot 3) + 3C + 2C + 1C$

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

$n - 2k = 0 \Rightarrow k = n/2$

$T(n) = 2^{n/2}\, T(n - n) + (2^{n/2} - 1)c$

$T(n) = O(2^{n/2}) \approx O(2^n)$

Space complexity will be $O(n)$ for the recursion stack which go to size n in worst case.

13) a) $n \log n$

Program:

```
for (int i = 0; i < n; i++)
{
    for (int j = 1; j < n; j *= 2)
    {
        cout << i << j;
    }
}
```

b) n^3

program:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            cout << i << j << k;
```

c) $\log(\log(n))$

```
void fun(int n)
{
    int c = 0;
    while (n > 0)
    {
        c++;
        n /= 2;
    }
}
:
int x = fun(n)
for (int i = 1; i <= x; i = i * 2)
    cout << i << x;
```

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

we can assume $T(n/2) \geq T(n/4)$

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

using master's theorem

$a = 2, \; b = 2$.

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

$f(n) = cn^2$

$n^k = n^2$

$k = 2$.

$\because \log_b a < k$.

$1 < 2$

Complexity $= O(n^k)$

$= O(n^2)$

15) $\because$ Inner loop will run n/i times.

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

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

$= n\log n$

$O(n\log n)$

16) Assuming pow(i, k) works in log(k) time.

we can express the runtime as.

$$\sqrt[k]{\sqrt[k]{\sqrt[k]{\cdots\sqrt[k]{n}}}} <= \sqrt[k]{2}$$

,

$$n^{\frac{1}{k^m}} \leq 2^{\frac{1}{k}}$$

raise both sides to k^m

$$n^{k^m/k^m} \leq 2^{\frac{k^m}{k}}$$

$$n \leq 2^{k^{m-1}}$$

$\log(n) \leq k^{m-1} \underbrace{\log(2)}$ ___ constant

$\log(\log(n)) \leq (m-1) \underbrace{\log k}$ ___ constant

$\log(\log(n)) + 1 \leq m$

$\because$ pow(i, k) takes $\log(k)$ time.

Complexity $= O(\log(k) \cdot \log(\log(n)))$

Q 18)

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

b) $1 < \cancel{2}n < 2n < 4n < \log(\log n) < \log(\sqrt{n}) < \log(n) < \log(2n)$
$< 2\log(n) < \log(n!) < n\log(n) < n^2 < 2^n \cdot 2 < n!$

c) $96 < \log_8(n) < \log_2(n) < n\log_6 n < n\log_2 n < \log(n!) < 5n$
$< 8n^2 < 7n^3 < 8^{2n} < n!$

Q19)
```
for(int i=0; i<n ; i++)
{  if (arr[i] is equal to key)
      print index and break
   else
        continue
}
```

Q20) Iterative:
```
void insertionSort(vector<int> &arr)
{
    int n = arr.size();
    for(int i = 0; i<n; i++)
    {
        int j = i;
        while(j>0 && arr[j] < arr[j-1])
        {
            swap(arr[j], arr[j-1]);
            j--;
        }
```

```
    }
}
```

Recursive:

```
void insertionSort (vector <int> &arr, int i)
{
    if (i<=0) return;
    insertionSort (arr, i-1);
    int j=i;
    while (j>0 && arr[j] < arr[j-1])
    { swap (arr[j], arr[j-1]);
      j--;
    }
}
```

It is called online sorting algorithm because it does not have the constraint of having the entire input available at the begining like sorting algorithms as bubble sort or selection sort. It can handle data piece by piece.

21) Quicksort: $O(n \log n)$
Mergesort: $O(n \log n)$
Bubble sort: $O(n^2)$
Selection sort: $O(n^2)$
Insertion sort: $O(n^2)$

22) Inplace: Bubble sort, selection sort, quicksort, insertion sort
Stable: bubble sort, insertion sort, merge sort
Online: insertion sort

23) Iterative:

```
int low = 0, high = n-1
while ( low<=high )
{
    mid = ( low+ high)/2
    if ( key = = a[mid])
        print mid & break;
    if ( key > a[mid])
        low = mid + 1;
    else high = mid - 1;
}
```

Recursive:

```
int BS( arr, low, high, key)
{
    if ( low > high)
        return -1;
    mid = (low + high) /2 ;
    if (arr[mid] = = key) return mid ;
    if (arr[mid] > key) return BS (arr, low, mid - 1);
    else return BS( arr, mid +1, high );
}
```

→ Time complexity of binary search:

iterative = $O(\log n)$

recursive = $O(\log n)$

→ Space complexity of binary search.

iterative = $O(1)$

recursive = $O(\log n)$

→ Time complexity of linear search

iterative = $O(n)$

recursive = $O(n)$

Space complexity of Linear search.

iterative = $O(1)$

Recursive = $O(n)$

24) Recursive ~~iteration~~ relation for binary search

$$T(n) = T(n/2) + 1.$$