DAA Assignment 1.

1. What do you understand by Asymptotic Notations? Define different asymptotic notation with examples.

Ans: Asymptotic notations are used to write fastest and slowest possible running time for an algorithm These are also referred to as 'best case' and 'worst case' respectively.
Three types of asymptotic notations to represent the growth of any algorithm, as input increases

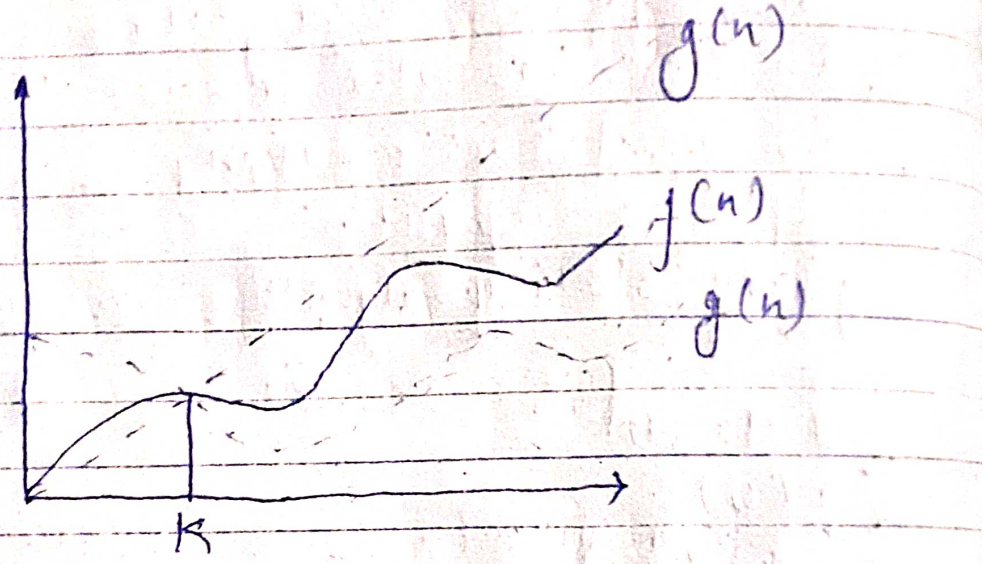
- Big Theta (Θ)
- Big O (O)
- Big Omega (Ω)

Big Theta Notation → The time complexity represented by the Big theta notation is like the average value or range b/w which the actual time of execution of algorithm will be.

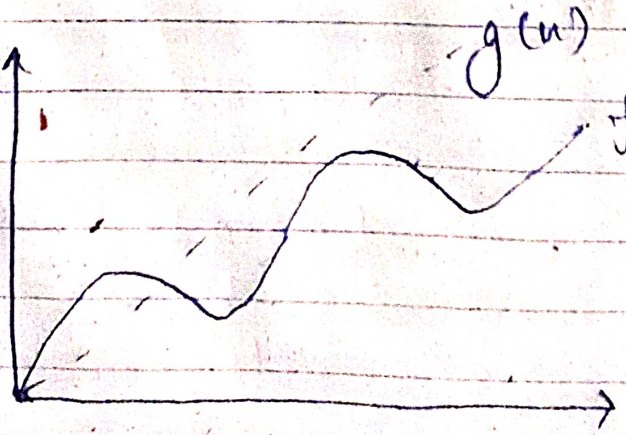
Eg. → $3n^2 + 5n$.

We use the Big θ notation to represent this, then the time complexity would be $\theta(n^2)$ ignoring the constant coefficient and ignoring insignificant part, which is $5n$.

$\theta(f(n)) = \{ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > no \}$.

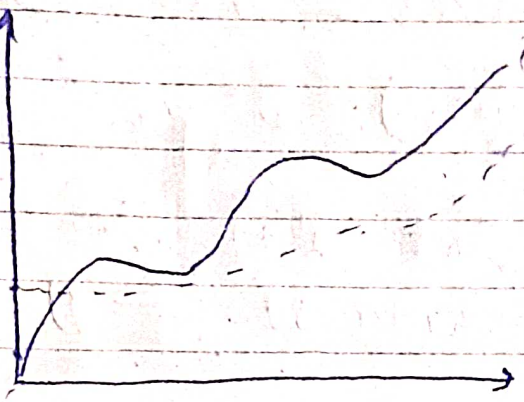


Big O Notation (O) → It is the formal way to express the upper bound of an algorithm running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete



Eg. → $O(f(n)) = \{ g(n)$: there exists $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n > n_0 \}$.

Omega Notation (Ω) → The notation Ω is the formal way to express the lower bound of an algorithm's running time. It measures the best case time an algorithm can possibly take to complete.



$\Omega(f(n)) \geq \{ g(n)$: there exists $c > 0$ and n_0 such that $g(n) < c \cdot f(n)$ for all $n > n_0 \}$

2. for ($i = 1$; $i < n$; $i = i * 2$)

1, 2, 3, 4, 8, - - - -

$T(n) = O(\log_2 n)$.

3. Let us solve this using substitution:

$$T(n) = 3T(n-1) \quad \text{— (1)}$$

Put $n = n-1$ in (1), we get

$$T(n-1) = 3T(n-1-1)$$
$$T(n-1) = 3T(n-2) \quad \text{— (2)}$$

Put value of $T(n-1)$ from (2) in (1), we get,

$$T(n) = 3(3T(n-2))$$
$$T(n) = 3^2(T(n-2)) \quad \text{— (3)}$$

Putting $n = n-2$ in (1), we get,

$$T(n-2) = 3T(n-2-1)$$
$$T(n-2) = 3T(n-3) \quad \text{— (4)}$$

Put the value of $T(n-2)$ from (4) in (3), we get,

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

So,

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

$T_n = 3^n$.

So, time complexity of this function is $O(3^n)$.

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

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

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

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

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

$\vdots$

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

$= 2^n(1) - 2^{n-1} - 2^{n-2} - \cdots - 2^2 - 2^1 - 2^0$.

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

$= 2^n - 2^n + 1$

$= 1$

Time complexity is $O(1)$.

5. We can define the terms 'S' according to relation, $S_i = S_{i+1} + 1$.

If k is total number of iterations taken by the program, then while loop terminates.

if $1 + 2 + 3 + \cdots + k$.

$= \left[\frac{k(k+1)}{2}\right] > n$.

so, $k = O(\sqrt{n})$

Time complexity of above function is $O(\sqrt{n})$.

6. If c is the total no. of iterations taken by program.

∴ then loop terminates,

$$(1)^2 + (2)^2 + (3)^2 + \dots + (\sqrt{n})^2$$

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

7. $T(n) = O(n * \log_2 n * \log_2 n)$.
$= O(n * (\log_2 n)^2)$.
$= O(n (\log_2 n)^2)$.

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

$T(n) = T(n-4) + n^2 + (n-1)^2$.
$T(n) = T(n-5) + n^2 + (n-1)^2 + (n-2)^2$.
⋮

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

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

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

$T(n) = T(1) + (4^2 + 5^2 + \dots n^2)$.

$T(n) = T(1) = \left(\frac{(n-3)(n-2)(2n-5)}{6}\right)$

$T(n) = 1 + \left(\frac{2n^3 + \dots}{6}\right)$

$T(n) = n^3$.

$T(n) = O(n^3)$.

9. $i = 1$ n times

$i = 2$ 1, 3, 5, - - - n/2

$i = 3$ 1, 4, 7 - - - n/3.

$i = n$ 0.

$T(n) = \left(n + \frac{n}{2} + \frac{n}{3} + \dots\right)$

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

10. For the relation, n^k and a^n, what is the relation

$k \geq 1$ and $a \geq 1$.

relation is n^k is $O(c^n)$.

11. 0, 3, 6, 10, 15, - - - n.

k^{th} term $= \frac{k(k+1)}{2} = \frac{k^2 + k}{2}$.

$k \simeq \sqrt{n}$ $T = \theta\sqrt{n}$.

12. Recurrence Relation of Fibonacci Series is

$T(n) = \{ T(n-1) + T(n-2) + 1 \}$.

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

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

$T(n) = 8T(n-6) + 7$.

$T(n) = 16T(n-8) + 15$.

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

for $T(n-2k) = T(0)$

$n = 2k$

$k = n/2$.

$T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)$

$= 2^n - 1$.

$= O(2^n)$.

Hence, space complexity of fibonacci series is O(n) as it depends on height of recurrence and it is equal to n in fibonacci series.

13. For n log n.

```
for (int i=0; j<n; i++)
{
    for (int=0; i<n; i=i*10)
    {
        printf ("hi");
    }
}
```

12. Recurrence Relation of Fibonacci Series is

$T(n) = \{ T(n-1) + T(n-2) + 1 \}$.

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

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

$T(n) = 8T(n-6) + 7$.

$T(n) = 16T(n-8) + 15$.

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

for $T(n-2k) = T(0)$

$n = 2k$

$k = n/2$.

$T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)$

$= 2^n - 1$.

$= O(2^n)$.

Hence, space complexity of fibonacci series is O(n) as it depends on height of recurrence and it is equal to n in fibonacci series.

13. For n log n.

```
for (int i=0 ; j<n ; i++)
{
    for (int=0 ; i<n ; i=i*0)
    {
        printf ("GG * 99");
    }
}
```

```
void main()
{
    fun();
}
```

for n^3,

```
#include<stdio.h>
void main()
{
  int n;
  cin>>n;
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<n; j++)
    {
      for(int k=0; k<n; k++)
      {
        cout<<k;
      }
    }
  }
}
```

for log (log n).

```
#include<bits/stdc++.h>
```

```
void main()
{
    fun();
}
```

for n^3,

```
#include<stdio.h>
void main()
{
  int n;
  cin>>n;
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<n; j++)
    {
      for(int k=0; k<n; k++)
      {
        cout<<k;
      }
    }
  }
}
```

for $\log(\log n)$.

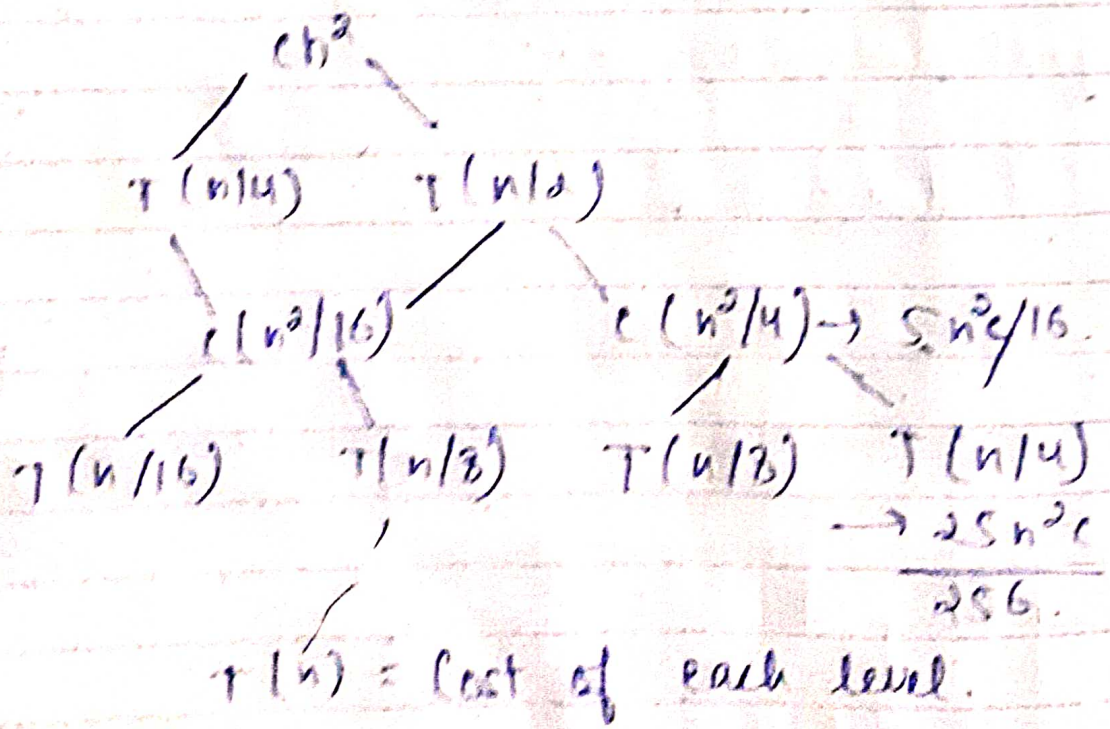
```
#include<bits/stdc++.h>
```

```
void fun(int n)
{
    if (n == 2)
        return 1;
    else
        fun(sqrt(n));
}
void main()
{
    fun(100);
}
```

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

$T(1) = 0$

$T(0) = 0$



$T(n)$ = cost of each level.

$T(n) = cn^2 + \frac{5cn^2}{16} + \frac{25cn^2}{256} + \ldots$

It is a G.P.
with a $= n^2$

$r = 5/16$.

So sum is S_∞

$T(n) = cn^2\left(\frac{1}{1-\frac{5}{16}}\right) = \frac{16}{11}cn^2$

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

15. $n, n/2, n/3, n/4, n/5 \ldots$

$k = \log_2 n$.

$n(1, 1/2, 1/3, 1/4, 1/5, \ldots 1/n)$

$(n(\log n))$.

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

16. $2, 2^k, 2^{k^2}, 2^{k^3}, \ldots$

It is a G.P.

where $a = 2$

$r = 2^k$.

k^{th} term $= ar^{k-1}$.

$n = 2(2^k)^{k-1}$.

Let $k^{k-1} = x$.

$k \log_2 k = \log x$.

$k = \log x$ — ①

$$n = 2^x$$

$$\log_2 n = x \log_2 2.$$

$$x = \log_2 n.$$

$$\log x = \log(\log n).$$

From (1),

$$k = \log(\log(n)).$$

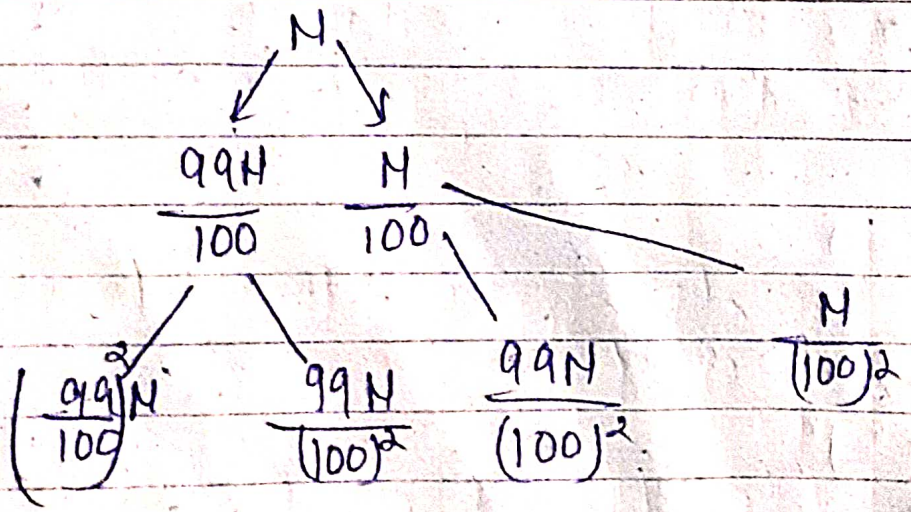
$$T(n) = O(\log(\log(n))).$$

17. Pivot is divided in 99% and 1%.

So,

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

Now, so here we can use a extreme of a tree where starting point is N.



$$N\left(\frac{(99)(99)}{(100)(100)} + \frac{(99)(1)}{(100)(100)}\right) + \frac{100\,N}{100\times100}$$

$$= \frac{99N}{100} + \frac{N}{100}$$

$$= N.$$

So, cost of each level is N rs.

Total cost = height * cost of each level

So for first stream = $N, \frac{99}{100}N, N\left(\frac{99}{100}\right)^2 N, \ldots$

$$\left(\frac{99}{100}\right)^{n-1} N = 1.$$

$$\left(\frac{99}{100}\right)^{n-1} = \frac{1}{N}$$

$$N = \left(\frac{100}{99}\right)^{n-1}.$$

$$\text{Log } N = n \log(1)$$

$$n = \log N \text{ or.}$$

$$n = \frac{\text{Log } N}{\text{Log}\left(\frac{100}{9}\right)} + 1.$$

Height of second stream

$N, N/100, N/(100)^2, N/(100)^3 + \ldots$

$$N\left(\frac{1}{100}\right)^{n-1} = 1.$$

$$N = (100)^{n-1}.$$

$$(n-1)\log 100 = \log N.$$

$$n = \frac{\log N}{\log 100} + 1 \quad \text{or} \quad n = \log(N) \quad \text{(approx)}.$$

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

So time complexities $O(N \log N)$.

height of both extreme is $\frac{\log N}{\log 100} + \log \frac{1}{100}$.

and $\frac{\log N}{\log \left(\frac{100}{99}\right)} + \log \frac{99}{100}$

So, we conclude that if division is done more than height of tree will be more and when division ratio is less than height is less.

18. a) $O(100) < O(\log(\log n)) < O(\log(n)) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(2^{2n}) < O(4^n)$.

b) $O(1) < O(\log(\log(n))) < O(\log(n)) < O(\log 2n) < O(2 \log n) < O(n) < O(n \log(n)) < O(\log(n!)) < O(2n) < O(4n) < O(n^2) < O(n!) < O(2(2^n))$.

c) $O(96) < O(\log_8(n)) < O(\log_{11}(n)) < O(\log n) < O(n \log(n)) < O(n \log_2(n)) < O(5n) < O(8n^3) < O(7n^3) < O(n!) < O(8^* 2^n)$

19. void linear search (int arr[], int n)

```
{
  for(i=0 to i=n)
      if arr[i] == key
          cout << "found";
      else
          continue.
```

20. Iterative Insertion sort.

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

Recursive insertion sort.

```
If n<=1.
    return;
Insertion sort(arr, n-1);
last = arr[n-1];
 j = n-2
while(j>=0 and arr[j]>last)
{
    arr[j+1] = arr[j]
    j--
}
arr[j+1] = last.
```

Insertion sort is casual online sorting because it does not know ~~the~~ whole input, it might make decisions that later turn out to be ~~opt~~ not optimal.

21.

	Time Complexity			Space
	Best	Average	Worst	
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

22

	Inplace	Stable	Online Sorting
Bubble	Yes	Yes	No
Selection	Yes	No	No
Insertion	Yes	Yes	Yes
Merge	No	Yes	No
Quick	Yes	No	No
Heap	Yes	NO	No

23. Binary search (arr, int n, key)

```
{
    beg = 0
    end = n-1
    while (beg <= end)
        mid = (beg + end)/2
        if (arr[mid] == key)
            found
        else if (arr[mid] < key.
            beg = mid + 1
        else
            end = mid - 1.
}
```

Time complexity of linear search = O(n)
Space complexity of linear search = O(1)
Time complexity of binary search = O(log n)
Space complexity of binary search = O(n)

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