DAA Assignment - 1

- Submitted by
Kumar Harsh
Section : SE
Roll no. : 2015078

Ans 1 → Asymptotic notations → To analyze an algorithm running time identifying its behaviour as the input size for the algorithm increases.

Types of asymptotic notations

i) Big - Oh (O)

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

if

$f(n) \leq c.g(n)$

$\forall\ n \geq n_0$, some constant $c > 0$

ii) Big Omega (Ω)

$f(n) = \Omega(g(n))$ if $f(n) \geq c.g(n)$

$\forall\ n \geq n_0$, some constant $c > 0$

iii) Theta (Θ)

$f(n) = \Theta(g(n))$ if $c_1(g(n)) \leq f(n) \leq c_2(g(n))$

$\forall\ n \geq \max(n_1, n_2)$

iv) Small - Oh (o)

$f(n) = o(g(n))$

$f(n) < g(n)\ \forall\ n > n_0$ and $\forall\ c > 0$

$n = o(n^2)$

V) Small omega (ω)

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

$f(n) > c \cdot g(n)$ $\forall$ $n > n_0$ and $\forall$ $c > 0$

$n^2 = \omega(n)$

Ans 2 → $i = 1, 2, 4, 8, \ldots n$

$2^0, 2^1, 2^2, 2^3 \ldots 2^k$

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

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

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

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

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

$\Rightarrow k = 1 + \log_2(n)$

$\therefore$ Time Complexity $= O(\log_2 n)$.

Ans 3 → $T(n) = \{ 3T(n-1)$ if $n > 0$, otherwise $1 \}$

$T(n) = 3T(n-1)$ —— 1

$T(1) = 1$

put $n = n-1$ in eqn 1

$T(n-1) = 3T(n-2)$ —— 2

put value of $T(n-1)$ from 2 to 1

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

$T(n) = 9T(n-2)$ —— 3

put $n = n-2$ in eqn 1

$T(n-2) = 3T(n-3)$ —— 4

put $T(n-2)$ from eqn 4 to eqn

$T(n) = 9(3T(n-3))$

$T(n) = 27\,T(n-3)$

$T(n)$

⋮

$T(n) = 3^K T(n-K)$ —— (5)

$T(1) = 1$

$n - K = 1$

$K = n - 1$

put value of K in eqn (5)

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

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

$\therefore T(n) = O(3^n)$

Ans 4 → $T(n) = 2T(n-1) - 1$

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

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

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

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

$\vdots$

$T(n) = 2^n (T(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0$

$= 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^0$

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

$= 1$

$\therefore T(n) = O(1)$

Ans 5 → $1, 3, 6, \ldots K \leq n$

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

$O(K^2) = n$

$K = \sqrt[2]{n}$

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

Ans 6 → void function (int n)

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

$$\underbrace{\text{count++;}}_{n}$$

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

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

$= 3 + n^2 + 4n$

$= n^2$

$\therefore$ T.C $= O(n^2)$

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++;
}
```

i	j	k
$\frac{n}{2}$	$\left(\frac{n}{2}\right)$ log n	$\log n \left(\frac{n}{2}\right)$
,	,	,
,	,	,
,		,

$\cancel{\log}$ $(\log n) * \log(n)$

$\therefore$ T.C $= O(\log^2 n)$

Ans 8 $\rightarrow$ function(int n){
if(n==1)
return; $\rightarrow 1$
for(i=1 to n)
{ for(j=1 to n) $\Big]\ n*n$
print("*"); $\rightarrow 1$
}
function(n-3); $\rightarrow n*n^2$
}

$\Rightarrow 1+n^2+1+n^3$
$= n^3+n^2+2 \Rightarrow O(n^3)$
$\therefore T.C = O(n^3)$

Ans 9 $\rightarrow$ void function(int n){
for(i=1 to n){
for(j=1;j<=n;j=j+1)
printf("*");
}
}

i	j	times
1	$1 \to n$	$\left(\frac{n+1}{2}\right)$
		⋮
		$\log n \frac{(n+1)}{2}$

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

Ans 10 → Asymptotic relation between n^k and a^n

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

$$n^k \leq C a^n$$

$$a^n + n^k \leq C a^n - a^n$$

$$a^n + n^k \leq a^n(C-1)$$

$$\frac{a^n + n^k}{a^n} \leq C-1$$

$$C \geq 1 + \frac{n_0^k}{a^{n_0}} + 1$$

$$C \geq 2 + \frac{n_0{}^k}{a^{n_0}}$$

let $K = 1$

$a = 1.5$

$$C \geq 2 + \frac{n_0^1}{1.5^n}$$

$$n_0 = 1$$

$$C \geq 2 + \frac{1}{1.5}$$

$$C \geq 3.0 + 1$$

$$C \geq 4$$

Ans 11 →
```
void fun (int n)
{ int j=1, i=0;
  while (i<n)
  { i=i+j;
    j++;
  }
}
```

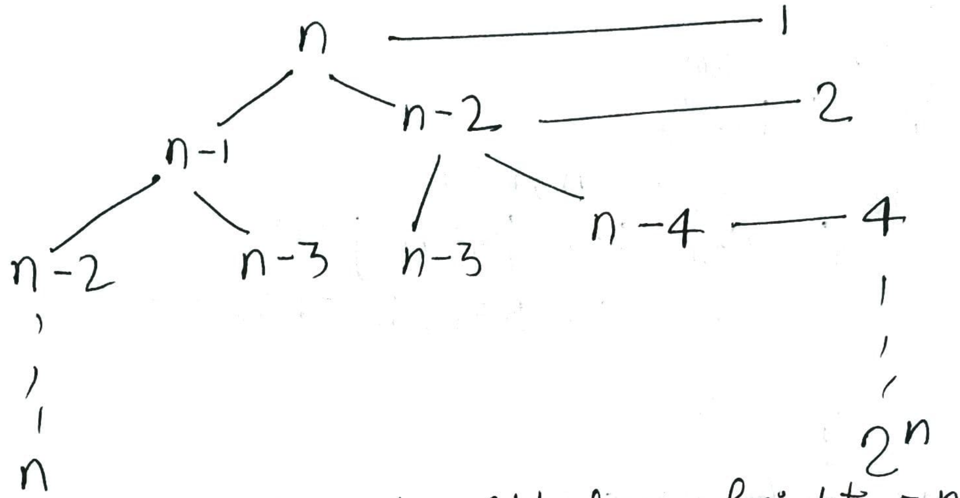
Time Complexity $= O(n)$

Here, while loop statement executes $(n-1)$ times, then 'i=i+j' operation n times execute and j++ also execute n times so the T.C is $O(n)$

Ans 12 → In fibonacci series, the first 2 no. are 0 and 1, and each subsequent no. is sum of previous two no. So recurrence relation is

$$f_n = f_{n-1} + f_{n-2}$$

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



Here longest branch will have height = n

$$T(n) = 1 + 2 + 4 + \cdots + 2^n$$

$$a = 1, \ r = 2$$

$$\frac{a(r^{term} - 1)}{r-1} = \frac{1(2^{n+1} - 1)}{2-1} = 2^{n+1} - 1$$

$$T(n) = O(2^{n+1}) = O(2^n \cdot 2^1) = O(2^n)$$

The space complexity of this program is $O(n)$ because the maximum no. of elements that can be present in the implicit function call stack i.e. the maximum depth is proportional to the n.

Ans 13 → O(n(logn)) code

```
int n;
for(int i=0; i<n; i++){
    for(int j=n; j>0; j/=2)
    { cout<<"Hi";
    }
}
```

→ O(n³) code

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

O(log(log n)) code

```
class Solution {
public: int cprime(int n) {
    if (n<2)
        return 0;
    boolean[] nprime = new boolean[n];
    nprime[1] = true;
    int numnprime = 1;
    for(int i=2; i<n; i++)
    { if(nprime[i])
        countinue;
      int j=i*2;
      while (j<n)
      { if (!nprime[j])
        { nprime[j]=true;
          numnprime ++;
        } j+=i;
      }
    } return(n-1) - numnprime;
}
```

Ans 14 → $T(n) = T(\frac{n}{4}) + T(\frac{n}{2}) + cn^2$

Solving by Recursion Tree Method

$T(1) = c$

$T(0) = 0$

$T(\frac{n}{4}) = \frac{c(n^2)}{16}$

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

cn^2 ——— 1

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

$T(\frac{n}{8})$ $T(\frac{n}{16})$ $T(\frac{n}{4})$ $T(\frac{n}{8})$ ——— ~~3~~4 — $T(\frac{n}{16})$

$T(\frac{n}{16})$ $T(\frac{n}{32})$ $T(\frac{n}{32})$ $T(\frac{n}{64})$ $T(\frac{n}{8})$ $T(\frac{n}{16})$ $T(\frac{n}{32})$

So here,

$$T(n) = c\left(n^2 + 5\left(\frac{n^2}{16}\right) + 25\left(\frac{n^2}{256}\right) + \cdots\right)$$

$$\text{ratio} = \frac{5}{16}$$

$$= \frac{n^2}{(1 - 5/16)} = O(n^2)$$

Ans 15 →

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

i	j	times
1	$1 \to n$	n
2	$1 \to n$	$n/2$
⋮	⋮	⋮
n		n/n

$T(n) = n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n}$

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

$T(n) = n \log n$

Ans 16 → for(int i=2; i<=n; i = pow(i,k))
{ // some O(1) expressions
}

Here $i = 2, 2^c, 2^{c^2}, 2^{c^3} \ldots 2^{c^{\log_c \log(n)}}$

The last term has to be $<= n$

$= 2^{c^{\log_c(\log(n))}}$ ~~2log~~

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

There are in total $\log_c(\log(n))$ iterations & each table takes a constant amount of time to run, total time complexity

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

Ans 19 → Linear search Pseudocode for search an element in sorted array

```
void LinearSearch(int *arr, int n, int key)
{   for(int i=0; i<n; i++)
    {  if (arr[i]<=key)
       {  if (arr[i]==key)
             return i;
       }
    }
    return -1;
}
```

Ans 20 → Recursive Insertion Sort

```
void RecursiveInsertionSort(int *arr, int i, int n)
{   int value = arr[i], j=i;
    while (j>0 && arr[j-1] > value)
    {     arr[j]=arr[j-1];
          j--;

    }
    arr[j] = value;
    if(i+1<=n)
    {   RecursiveInsertionSort(arr, i+1, n);

    }
}
```

Iterative Insertion Sort

```
void IterativeInsertionSort(int *arr)
{   for(int i=1; i<=arr.length(); i++)
    {      value = arr[i];
           j = i;
       while (j>0 && arr[j-1] > value)
       { arr[j] = arr[j-1];
                j--;

       }
       arr[j] = value;

    }

}
```

Insertion Sort is an online algorithm because an online algorithm doesnot know the whole input it might take decisions that later turn out not to be optimal. Insertion sort produces optimum result.

The sorting algorithms which are discussed in lectures are -

Selection sort Algorithm

It sorts an array by repeatedly finding minimum element from the unsorted part & putting it at the beginning.

Algorithm

```
void SelectionSort(int *arr, int n) {
        int i, j, temp, min;
        for i <- 0 to n-1
        { min = i;
            for j <- i+1 to n
            if(arr[j] < arr[min])
                min = j;
            temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
}
```

Bubble Sort Algorithm

It works by repeatedly swapping adjacent elements if they are in the wrong order.

Algorithm

```
BubbleSort(int arr[], n)
{   int Swap, i, j;
    for i ← 0 to n
    { swap = 0;
      for j ← 0 to n-i-1
      { if(arr[j] > arr[j+1])
        { swap (arr[j], arr[j-1]);
          swap = 1;
        }
      }
      if (swap == 0)
          break;
    }
}
```

~~Ans Q2~~

Ans 21 → Complexity of Bubble Sort

$$TC = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-i-1} 1 \right)$$

$$= \sum_{i=0}^{n-1} (n-i) = n + (n+1) + \dots + 1$$

$$= n * \frac{(n+1)}{2} = O(n^2)$$

Best case $= O(n^2)$

Average case $= O(n^2)$

Worst case $= O(n^2)$

Space complexity $= O(1)$

Complexity of Selection Sort

T.C = $O(n^2)$
B.C = $O(n^2)$
Av. C = $O(n^2)$
W.C = $O(n^2)$
S.C = $O(1)$

Complexity of Insertion Sort

T.C = $O(n)$

B.C = $O(n)$
Av. C = $O(n^2)$
W.C = $O(n^2)$
S.C = $O(1)$

Ans 22 →

in-place algorithms	Stable Algorithms	Online Algorithms
→ Bubble sort → insertion sort → Selection sort → quick Sort → Heap Sort	→ Bubble Sort → insertion sort → Merge sort	→ insertion sort

Ans 23 → Recursive Binary Search Pseudocode

```
int binarysearch(int arr[], int l, int r, int n)
{   if( r>=l)
    int mid ← (l+r)/2;
    if(arr[mid] = n)
        return mid;
```

```
else if( arr[mid] > x )
      return binarysearch(arr, l, mid-1, x);
else
      return binarysearch(arr, mid+1, r, x);

}
return -1;

}
```

Iterative binary Search Pseudocode

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

Space complexity of binary Search

Best S.C = $O(1)$ iterative

Avg. S.C = $O(\log n)$ recursive

Time Complexity of binary Search

Best Case = $O(1)$

Avg. Case = $O(\log_2 n)$

Worst Case = $O(\log_2 n)$

Spare Complexity of linear Search = O(1)

Time " " " → B.C = O(1)

A.C and W.C = O(n)

<u>Ans 24</u> → Recurrence Relation for binary search

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

<u>Ans 18</u> → a) 100 < loglogn < logn < log(n!) < root(n) < n < nlogn < 2^n < 4n < 22^n < n!

b) log(log(n)) < $\sqrt{\log n}$ < log(n) < 2 log(n) < log(2n) < log(n!) < n < 2n < 4n < nlogn < n^2 < $2(2^n)$ < n!

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