**Design and Analysis of Algorithms**

**Frequency Count Method**

Ex - 1 :- Algorithm sum(A , n)

Input : A – array of n elements

1 . s <- 0 ----------- 1 time

2 . for(i=0 ; i < n ; i++) -------- (i=0) -> 1 time , (i<n) -> n + 1 times and (i++) -> n times , But we will consider only (n+1)times

3 . s <- s + A[i] ---------- n times

4 .end for

5 .return s ------ 1 time

Analyzation :-

**TIME COMPLEXITY**

-Each statement will take 1 unit of time .

Hence , total unit of time taken = 1 + (n+1) + n + 1 = (2n + 3) unit of time , OR

Time-function , f(n) = 2n + 3 , since the degree of the polynomial is 1

hence , ***time-complexity = O(n)***

**SPACE COMPLEXITY**

-Each variable will take 1 word of memory

Hence , A takes n words , and

n , s , i will take 1 word each

hence , total space , S(n) = n+3 , since its degree is also 1

***space-complexity = O(n)***

Ex – 2 : - Analyze the time and space complexity of an algorithm of sum of two n\*n matrices

Algorithm add(A,B,n)

Input : A , B – Two square matrices with n\*n dimensions

1 . for(i=0;i<n;i++) ------------ (n+1) times

2 . for(j=0;j<n;j++) ---------- n \* (n+1) times

3 . C[i,j] = A[i,j] + B[i,j] ------ n \* n times

4 . end for

5 . end for

**TIME COMPLEXITY**

Hence , Time function f(n) = 2n2 + 2n + 1 , since degree = 2(since the highest order-term is 2n2)

***Time-complexity = O(n2)***

**SPACE COMPLEXITY**

A , B , C - n\*n words each ; i , j , n – 1 word each

Hence , Space function S(n) = 3 n2+ 3 , degree = 2

***Space – complexity = O(n2)***

Ex – 3 :- Analyze the algorithm of multiplication of two square matrices of n\*n dimension .

Algorithm product(A,B,n)

1 . for(i=0;i<n;i++) --------- (n+1) times

2 . for(j=0;j<n;j++) -------(n+1)\*n times

3 . C[i,j] = 0 ; --------n \* n times

4 . for(k=0;k<n;k++) -------n\*n\*(n+1) times

5 . C[i,j] = C[i,j] + A[i,k]\*B[k,j] -----------n\*n\*n times

6 . end for

7 . end for

8 . end for

Time-function f(n) = 2 n3 + 3 n2 + 2n + 1 , degree = 3

***Time-complexity = O(n3)***

Space-function S(n) = 3 n2 + 4 , degree = 2

***Space – complexity = O(n2)***

Ex – 4 :-

1 . for(i=0;i<n;i++) ----------(n+1) times

2 . for(j=0;j<i;j++) -----------{n\*(n+1)/2 + 1} times

3 . stataement . ----------------n\*(n+1)/2 = **(n2 + n)/2**

4. end for

5 . end for

***Time-complexity = O(n2)***

Ex – 5 :-

1 . p <- 0

2 . for(i=1 , p<=n , i++)

3 . p = p + i

4 . end for

Assume that p becomes greater than n i.e **p>n** after **k times** , hence , the statement will run k times , Hence , *Time-complexity = O(k)*

then after k times , p = 1 + 2 + 3 + ….+ k = k\*(k+1)/2 = **(k2  + k)/2**

* P > n
* K^2 > n
* K = (n1/2)

Hence ***, Time-complexity = O(n1/2)***

Ex – 6 :-

1 . for(i=1 , i<n , i=i\*2)

2 . statement

3 . end for

Suppose the loop runs **k** times to stop(i.e **i>n**) , hence , **i = 2k** and time-complexity = O(k)

At time k **, i > n => 2k > n => k = log(n2) ,** Hence ***Time-complexity = O(logn2)***

* *In Algorithm analysis we take* ***ceil value*** *of log , i.e for decimal values we take the next largest integer ex:– 3.2 = 4*

Ex- 7 :-

1 . for(i=n , i>=1 , I = i/2)

2 . statement

3 . end for

=> Assume after **k** times loop stops i.e **i<1** , and **i = n/2k-1** , then , *time-complexity = O(k)*

=> **i < 1** => **n/2k-1 = 1** => **n = 2k-1** **=> logn = (k-1)log2** => **k-1 = log n2** => **k = log n2** (discarded 1 from RHS)

Hence ***, Time-complexity = O(log n)***

Ex – 8:-

1 . p = 0

2 . for(i=1,i<n,i=i\*2) ------- (**log n2**) times

3 . p++ ---------( **log n2**) times , hence , **p = log n2**

4 . end for

5 . for(j=1,j<p,j=j\*2) ------ (log p) times or **(log (log n2))** times

6 . statement ------- (log p) times or **(log (log n2))** times

7 . end for

Hence time-function , **F(n) = 2 (log (log n2)) + 2 log n2**

***Time-complexity = O(log (*log n2*))***

Ex-9 :-

1 .for(i=0 ; i < n ; i++) ---------- n times

2 . for(j = 1 , j<n , j=j\*2) --------(n\* **log n2**) times

3 . statement ---------- (n\* **log n2**) times

4 . end for

5 . end for

**{We are discarding the constant parts of time functions in each step for our convenience as only highest order terms decide the time-complexity}**

Time function ***, f(n) = 2(n\** log n2*) + n*** , since the highest order term is n\*logn , hence ,

***Time-complexity = O(n\** log n2*)***

**General observed notions for time-complexities :-**

1 . for(i=0 , i<n , i++) ----***n times , O(n)***

2 . for(i=0 , i<n , i = i+2) ------***n/2 times , O(n)***

3 . for(i=n , i>1 , i--) --------- ***n times , O(n)***

4 . for(i=1 , I <n , i=i\*2) ------ ***log2n times , O(log2n)***

5 . for(i=n , i>1 , i=i/2) ----- ***log2n times , O(log2n)***

6 . for(i=1 , i<n , i=i\*3) ----- ***log3n times , O(log3n)***

7 . for(i=n , i>1 , i=i/3) ----- ***log3n times , O(log3n)***