Computing Space Complexity

→ The space needed by an algorithm is the sum of the following components.

1) A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs.

This part typically includes the instruction space (i.e., space for the program code), space for simple variables and fixed-size component variables and constants [static data]

2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved [Dynamic memory Allocation]

The space requirement S(P) of any algorithm P may be written as

$$S(P) = C + S_p(\text{instance characteristics})$$

C: space

where C is a constant

→ When analyzing the space complexity of an algorithm, we concentrate solely on estimating S_p(instance characteristics)

We compute the space complexity interms of no. of memory words. A memory word is large enough (for e.g. 64 bits) to store any value (int, long int, float, double).

Ex 1

```
Algorithm  abc (a,b,c)
{
      return  a+b+b*c+(a+b-c)/(a+b) + 4.0;
}
```

The space needed for the above algorithm is

4 memory words

(One each for a, b, c and result)

The space needed by abc(a,b,c) algorithm is independent of instance characteristics.

$S(abc) = C + S_{abc}$ (instance characteristics)

$S(abc) = 4 + 0$

$S(abc) =$ 4 memory words

Example 2 :- Iterative function for sum

// To find sum of the n elements of array a.

```
Algorithm sum(a,n)
{
    s := 0.0;
    for I := 1 to n do
    {
        s := s + a[i];
    }
    return s;
}
```

Space needed is

one word for s

one word to store the value of i

one word for n

n words for storing n elements of array a.

$S(sum) = C + S_{sum}$ (instance characteristics)

$S(sum) = 3 + n$

$= n + 3$ words (memory words)

Example 3 : Recursive function for Sum

```
Algorithm Rsum(a,n)
{
    if(n≤0) then return 0.0;
    else
     return Rsum(a, n-1) + a[n];
}
```

→ The recursion stack space includes space for the formal parameters, the local variables and the return address

→ Each call for ~~storing~~ ~~n value~~ Rsum requires 3 words.

1 word for storing n value.
1 word for storing pointer a[]
1 word for storing return address

} Rsum(a, n-1]

Rsum function will be ~~called~~ (n+1) times.

Rsum(a,n)
↓
Rsum(a, n-1) + a[n]
↓
Rsum(a, n-2) + a[n-1]
⋮
Rsum(a, 1) + a[2]
↓
Rsum(a,0) + a[1]
↓
0.0 → sum is zero. Since array size = 0

∴ The recursion stack space needed
= 3(n+1)

Computing Time complexity of an algorithm :-

The time T(P) taken by a program P is the sum of the compile time and run (or execution) time. The compile time does not depend on the instance characteristics. We assume that a compiled program will be run (executed) several times without recompilation. We concern ourselves with just the runtime of a program. The runtime ~~of~~ is denoted by t_p (instance characteristics)

→ A program step is defined as a syntactically or semantically meaningful segment of a program that has an execution time (takes some CPU time for execution)

→ Step Counts To determine the no. of steps needed by a program, we introduce a global variable count.

No. of executable statements.

Ex1:-

```
Algorithm sum(a,n)        // count = 0
{
    s := 0.0 ;            // count := count + 1   → 1 step
    for i := 1 to n do    // count := count + 1   (n+1)
    {
        s := s + a[i] ;   // count := count + 1   → n times
                                                  → n steps
    }
    return s ;            // count := count + 1   → 1 step.
}
```

Total no. of steps required for algorithm

$= 1 + (n+1) + n + 1$

$= 2n + 3$

Ex 2:-

```
Algorithm Rsum(a,n)       // count := 0
{
    if (n ≤ 0) then       // count := count + 1
    {
        return 0.0 ;      // count := count + 1
    }
    else
    {
        return Rsum(a, n-1) + a[n] ;
    }
}
```

Rsum(a, n-1) → 1 + t Rsum(a, n-1)

Step for adding a[n], function calling, and return.

When analyzing a recursive program for its step count, we obtain a recursive formula.

$t_{RSum}(n)$ = { 2 if $n \le 0$; $2 + t_{RSum}(n-1)$ if $n > 0$ }

$t_{RSum}(0) = 2$

These recursive formulas are referred to as recurrence relations. We derive the total step count by using recursive substitution.

$t_{RSum}(n) = 2 + t_{RSum}(n-1)$ → 1st step of deviation

$= 2 + 2 + t_{RSum}(n-2) = 2 \times 2 + t_{RSum}(n-2)$ — 2nd step

$= 2 * 2 + 2 + t_{RSum}(n-3) = 3 \times 2 + t_{RSum}(n-3)$ — 3rd step

Similarly at n^{th} step, we can write

$= n \times 2 + t_{RSum}(n-n)$

$= 2n + t_{RSum}(0)$

$t_{RSum}(n) = 2n + 2$

The runtime is proportional to n.

It grows linearly with n.

Eg 3:

```
Algorithm matrixadd(a,b,c,m,n)
{
  for i:=1 to m do ---- (m+1) times → (m+1) steps
  {
    for j:=1 to n do ---- m(n+1)
    {
      c[i,j]:= a[i,j] + b[i,j]; ---- mn.
    }
  }
}
```

$C = A + B$

$$C = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m \times n} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}_{m \times n}$$

Total steps required

$$T(\text{matrixadd}) = (m+1) + m(n+1) + mn$$

$$= m + 1 + mn + m + mn$$

$$= 2mn + 2m + 1$$

$T(\text{matrixadd}) \propto mn$

$$T(\text{matrixadd}) = O(mn)$$

if $m = n$, i.e., matrices having same no. of rows and columns. Then time complexity of matrix addition

$$T(\text{matrixadd}) = O(n^2)$$

Time complexity of matrix multiplication

```
Algorithm matrixmul(a, b, c, n, n)
{
    for i:=1 to n do ⟶ (n+1)
    {
        for j:=1 to n do ⟶ n(n+1)
        {
            for k:=1 to n do ⟶ n*n(n+1)
            {
                c[i,j]:= c[i,j] + a[i,k] * b[k,j]; ⟶ n*n*n
            }
        }
    }
}
```

Total no. of steps required

$$T(matrixmul) = (n+1) + n(n+1) + n*n*(n+1) + n*n*n$$

$$= n+1 + n^2 + n + n^3 + n^2 + n^3$$

$$T(matrixmul) = 2n^3 + 2n^2 + 2n + 1$$

$$T(matrixmul) \propto n^3$$

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

Step Table method :-

The 2nd method to determine the step count of an algorithm is to build a step table, in which we list the no. of steps contributed by each statement (instruction).

s/e ⟶ Steps per execution no. of steps involved in an instruction.

Frequency → total no. of times each statement will be executed.

Eg 1 :-

Statement	s/e	Frequency	Total steps
1. Algorithm Sum(a,n)	0	–	0
2. {	0	–	0
3. s := 0.0;	1	1	1
4 for i := 1 to n do	1	n+1	n+1
5 {	0	–	0
6 s := s + a[i];	1	n	n
7. }	0	–	0
8. return s;	1	1	1
9. }	0	–	0
			2n+3

statement	s/e	Frequency	Total steps.
Algorithm Add (a,b,c, m,n)	0	–	0
{	0	–	0
for i := 1 to m do	1	m+1	m+1
{	0	–	0
for j := 1 to n do	1	m(n+1)	mn+m
{	0	–	0
c[i,j] := a[i,j]	1	mn	mn
}	0	–	0
}	0	–	0
}	0	–	0

Total no. of steps required = $2mn + 2m + 1$