Time Complexity

Algorithm Analysis

Time Complexity

- Performance analysis of an algorithm is accomplished in terms of the number of elements that constitute the input.
- This is often referred to as instance characteristic.
- Performance analysis of an algorithm is accomplished by counting the total no of *Program Steps*.
- A Program Step is a semantically meaningful segment of a program whose execution time is independent of the instance characteristics.

Time Complexity

- A Program step is assumed to consume one unit of Processor time.
- Consider the following fragment of the algorithm:

```
1) sum:=0;
2) sum:=sum+num;
```

- The above fragment has 2 Program steps and takes 2 units of time.

Time Complexity

- Each iteration of a for loop is assumed to be a Program step.
- An additional step is assumed for the condition check that results in the termination of the for loop.

```
for i:=1 to n do    //n+1 steps
{
     sum:=sum+i;    //n steps
}
Total no of steps = 2n+1
```

Time Complexity

- One method to count the Program steps is to make the algorithm print the total number of Program steps.
- Introduce a global variable `count` in the algorithm that is initialized to zero.
- The statement that increments the `count` is introduced in the algorithm before every Program step.

Time Complexity

- By the end of the algorithm execution, the variable **`count`** holds the total number of program steps.
- If the value of the variable **`count`** is printed at the end, the total number of program steps will be printed.

Time Complexity

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

Time Complexity

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

```
Algorithm Sum(a, n)
{
    count:=count+1;// initialize
    s:=0.0;
    for i:= 1 to n do
    {
        count:=count+1;// each for iteration
        count:=count+1;// each sum step
        s:=s+a[i];
    }
    count:=count+1;//last for iteration
    count:=count+1;//return
    return s;
}
```

Time Complexity

```
Algorithm Sum(a, n)
{
   for i:= 1 to n do
      count:=count+2;
   count:=count+3;
}
```

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

Time Complexity

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
        for j := 1 to n do
            c[i, j] := a[i, j] + b[i, j];
}
```

Time Complexit

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
        for j := 1 to n do
            c[i, j] := a[i, j] + b[i, j];
}
```

Time Complexity

```
Algorithm Add(a,b,c,m,n )
{
    for i:= 1 to m do
    {
        count:=count+2;
        for j:= 1 to n do
            count:=count+2;
    }
    count:=count+1;
}
```

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

Time Complexity

```
Algorithm Sumd (  num )
// returns the image of
num
{
   sum:=0;
   while(num ≠ 0)
   {
      sum:=sum+num%10;
      num:= num/10;
   }
   return sum;
}
```

Time Complexity

- The other method is to calculate the steps for execution and the frequency of execution whose product will give the total number of steps.

```
for i:=1 to n do//1 step for execution,  freq of n+1
{
      sum:=sum+i; //1 step for execution, freq of n
}
Total no of steps = 2n+1
```

Time Complexity

No.	Algorithm	S/E	F	T

Total T(n)=2n+3

Time Complexity

```
Algorithm Add(a, b, c, m, n)
{
    for i := 1 to m do
        for j := 1 to n do
            c[i, j] := a[i, j] + b[i, j];
}
```

Time Complexity

Statement	s/e	frequency	total steps

Time Complexity

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

While preparing the step table, create separate columns of S/E, Frequency of execution and Total Program steps for base case

Time Complexity

Statement	s/e	Frequency n=0 n>0	Total Steps n=0 n>0

Time Complexity

$T(n) = 2 \; if \; n = 0$
$T(n) = 2 + T(n-1) \; if \; n > 0$

The recurrence can be solved to obtain an expression for $T(n)$

$$
\begin{aligned}
T(n) &= 2 + T(n-1) \\
&= 2 + 2 + T(n-2) \\
&= 2 + 2 + T(n-2) \\
&\quad \quad \ldots
\end{aligned}
$$

Time Complexity

$$
\begin{aligned}
T(n) &= 2 + T(n-1) \\
&= 2 + 2 + T(n-2) \\
&= 2 + 2 + T(n-2) \\
&\qquad \ldots \\
&= 2 + 2 + 2 + \cdots n\ times + T(n-n) \\
&= 2 + 2 + 2 + \cdots n\ times + T(0) \\
&= 2n + 2
\end{aligned}
$$

Time Complexity

```
Algorithm Fibonacci(n)
{
   if(n=0)
      return 0;
   else if(n=1)
      return 1;
    else
      return Fibonacci(n-1)+ Fibonacci(n-2);
}
```

Time Complexity

Statement	s/e	Frequency n=0	n=1	n>1	Frequency n=0	n=1	n>1
Algorithm Fib(n)							
{							
if(n==0)	**1**	**1**	**1**	**1**	**1**	**1**	**1**
return 0;	**1**	**1**			**1**		
else if(n=1)	**1**		**1**	**1**		**1**	**1**
return 1;	**1**		**1**			**1**	
else							
return Fib(n-1)+ Fib(n-2);	**1+T(n-1)+T(n-2)**			**1**			**1+T(n-1)+T(n-2)**
}							
Total					**2**	**3**	**3+T(n-1)+T(n-2)**

Time Complexity

$T(n) = 2 \ if \ n = 0$
$T(n) = 3 \ if \ n = 1$
$T(n) = 3 + T(n-1) + T(n-2) \ if \ n > 1$

The recurrence can be solved to obtain an expression for $T(n)$

Time Complexity

```
Algorithm Fibonacci(n)
// Compute the nth Fibonacci number.
{
    if (n ≤ 1) then
        write (n);
    else
    {
        fnm2 := 0; fnm1 := 1;
        for i := 2 to n do
        {
            fn := fnm1 + fnm2;
            fnm2 := fnm1; fnm1 := fn;
        }
        write (fn);
    }
}
```

Time Complexity

St.No.	$n \leq 1$			$n > 1$		
	S/E	**F**	**T**	**S/E**	**F**	**T**
4	1	1	1	1	1	1
5	1	1	1			
8				2	1	2
9				1	n	n
11				1	n-1	n-1
12				2	n-1	2n-2
14				1	1	1
Total			2			4n+1

$$\boldsymbol{T(n) = 4n + 1}$$