3

MORE EXAMPLES ABOUT THE ASYPTOTIC NOTATION AND COMPLEXITY

Example 1:

int sum= 0; 1

for (i=0; i<10; i++) 11

sum = sum +i ; 10

<u>**What is the time complexity?**</u>

Sol: T=22

Example 2:

int z; 1

for (i=0; i<10; i++) 11 for

(j=0; j<5; j++) 10 x 6 z= i*j;

10 x 5

<u>**What is the time complexity?**</u>

T= 12+10 x 6 + 10 x 5= 12+10(6+5)=122

Example 3:

input f; 1

for (i=0; i<5; i++) 6 if

(i==2) 5

printf(i); 1

What is the time complexity?

Sol:

T=13

Calculate the time for Case (FOR STATMENTS):

1:

1. For (i=0, i<n, i++) n+1

$\underbrace{0\text{------------ } n-1}_{n}$

let n=10

i< n
0<10

1<10 :

: 11, which

is n+1 :

9 < 10

10 < 10

Case 2:

2. for (i=0; i<=n; i++) 0 n+2

------- n
n+1+1

Case 3:

for (i=1; i<n; i++) n
$\underbrace{1\text{------------}n-1}$

n-1+**1 (failure)**

Case 4:

for (i =1; i<=n; i++) n+1

1------------n

n+1

Theory:

f(n)= am n^m + am -1 n^{m-1} + … + a1 n1 + a_0 f(n)

= O (n^m)

RECURSION

Recursion means calling a function in itself. If a function invokes itself, then the phenomenon is referred to as recursion. However, in order to generate an answer, a terminating condition is must. In order to understand the concept, let us take an example.

Ex 1: If the factorial of a number is to be calculated using the function fac(n) defined as follows:

$$\textbf{fac(n) = n} \times \textbf{fac(n-1)}$$

and fac(1) = 1, and if the value of n is 5, then the process of calculating fac(5) can be explained with the help of Fig. 3.1. fac(1) is calculated and its value is used to calculate fac(2), which in turn is used for calculating fac(3). fac(3) helps to calculate fac(4) and finally, fac(4) is used to calculate fac(5). As is evident from Fig. 3.1, recursion uses the principle of last in first out and hence requires a stack. One can also see that had there been no fac(1), the evaluation would not have been possible. This was the reason for stating that recursion requires a terminating condition also.

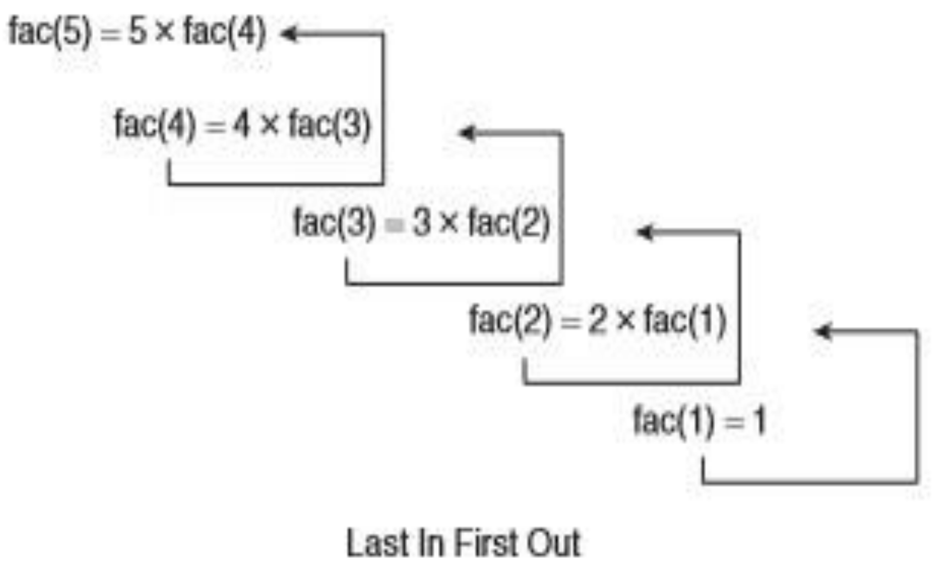

Figure 3.1 Calculation of factorial of 5

<u>Ex 2:</u> power

```
int power (int base, int exponent) {
if    (exponent    ==    0)    {
return 1;      } else {
        return base * power(base, exponent - 1);
    }
};
```

The key in any recursive function is the BASE CASE, without which the recursion will go on until it overflows the stack.

The exponent is passed in and immediately checked to see if it is zero. If it is, then 1 is returned, and the recursion terminated. At this point, the stack is cleared and final result returned.

Do a pencil check of the function to follow its flow:

Let the function be called with arguments 2 and 5 (base and exponent, respectively)

On paper we can begin to break down our running variables and track the interim returns:

```
base = 2 exponent
= 5
exponent !== 0 so base case is skipped
return 2 * power(2, 4) exponent = 4
return 2 * power(2, 3) exponent = 3
return 2 * power(2, 2) exponent = 2
return 2 * power(2, 1) exponent = 1
return 2 * power(2, 0) exponent = 0
return 1
```

Now that the base case is reached, the computer will have to cycle through all the returns that are currently stored internally in the stack.

```
return 1 * 2 = 2
return 2 * 2 = 4
return 4 * 2 = 8
return 8 * 2 = 16
return 16 * 2 = 32
```

Answer: 32

<u>Ex3:</u> sum=1+2+3+ …+ n

```
int sum(int n)
{
    if(n == 0)
return 0;     else
        return n + sum(n-1);
}
```