GROWTH OF FUNCTIONS

INTRODUCTION

In order to accomplish a task, the most important thing is to design a correct algorithm. An algorithm can be called correct if it accomplishes the required task. However, sometimes in spite of being correct, an algorithm may not be of much use, in the case where it takes a lot of time. For example, applying linear search in order to find out an element is correct, but what if the array contains more than 10^{10} elements? Even if one element is processed in 10^{-6} seconds, it will take 10,000 seconds or around 3 hours to search an element. Now imagine that the same task is to be accomplished in an array that contains the roll numbers of all the students of a university. In that case this procedure will require a lot of time. So, it is important that the algorithm should be correct as well as efficient. The understanding of running time is also important in order to compare the efficiency of two algorithms.

It is difficult to find the exact running time of an algorithm. It requires rigorous mathematical analysis. The calculation of exact running time also requires the knowledge of sequences and series and logarithms among others. Moreover, the exact analysis provides no additional advantage compared to an approximate analysis. The exact analysis gives the exact polynomial function that relates the input size with the running time, whereas the approximate analysis gives the power of input size on which the running time depends. For example, the exact running time of an algorithm may be $3 \times n^2 + 2 \times n + 3$. In this case, the approximate running time would be $f(n^2)$. So, the highest power of n is what matters while calculating the approximate running time of an algorithm. Even the constants that are there with the term containing the highest power do not matter. It may also be stated that the number of inputs to an algorithm may not always be the number of variables that are given as an input to the algorithm. For example, if an algorithm takes an array as an input, then the input size is generally taken as n and not 1. So, the idea is that since an array contains n elements, the number of inputs to the algorithm must be taken as the number of elements in that array. The algorithm will most probably deal with most, if not all, of the elements of the array.

The argument can be extended to a two-dimensional array as well. The number of inputs of an algorithm that manipulates an array having n rows and m columns is taken as $n \times m$, and not 1. This is because the number of elements in the data structure is $n \times m$.

1. BASIC MATHEMATICAL CONCEPTS

The definition of the general sequence and the sum of n terms of arithmetic, and geometric progressions have been dealt with in the present section. This lecture also throws light on logarithms, so that the idea of complexity can be understood clearly.

- **COMPLEXITY**

In examining algorithm efficiency we must understand the idea of complexity

a. Space complexity

b. Time Complexity

A- SPACE COMPLEXITY

- When memory was expensive we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)

S(P)= Const + Sp

Ex: What is the space complexity of the following code?

```
Float   abc (float a; float b , float c)
 {
    Return (a+b*6*c+(a+b+c)/(a+4.0)
}      Five space  ( a , b, c , return address , abc)
Sabc (a,b,c) = 0      Const
```

B- TIME COMPLEXITY

- Is the algorithm “fast enough” for my needs
- How much longer will the algorithm take if I increase the amount of data it must process

- Given a set of algorithms that accomplish the same thing, which is the right one to choose
- Time efficiency depends on :
  - ✓ size of input
  - ✓ speed of machine
  - ✓ quality of source code
  - ✓ quality of compiler

Time Complexity:

T(P)= Const + t_p

Where Const : compiler time and t_p : time of running program

Ex: What is the Time complexity of the following code ?

```
Float   abc (float a; float b , float c)                FC
 {
    Return (a+b*6*c+(a+b+c)/(a+)+4.0)                   1
 }
```

Steps count or FC = 1

T_{abc} (a,b,c) = 0 Const

Example :

$S = \sum_{i=0}^{n}$ a[i] Where i=0,1,2,….n

```
                                                        Fc
float     Sum (float a[], int n)
{
float s=0.0;                                            1
for(int i=0 ; i< n ; i++)                               n+1
```

```
 s+= a[i];                                   n
 return s;                                   1
 }                                       Fc= 2n+3
```

Tsum (n) = 2n+3

Ssum >= 4

Ssum=0

Space complexity :

$S_{rsum}(n)= 3(n+1)$

Time Complexity :

$$T_{rsum}(n)\sum_{2+trsum(n-1)}^{2} \quad \begin{matrix} if\ n\leq 0 \\ if\ n>o \end{matrix}$$

Sol

$t_{rsum}(n)= 2+ t_{rsum}(n-1)$

$= 2+2 +t_{rsum}(n-2)$

$= 2(2) + 2+ t_{rsum}(n-3)$

$= 3(2) + 2+ t_{rsum}(n-4)$

$= k(2) + t_{rsum}(n-k)$

Let k = n

$= n(2) + t_{rsum}(n-n)$

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

$= n(2) + 2$

$= 2n + 2$

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

$= O(n)$

Example: What is the Time complexity of the following code?

Let n=m FC

```
void  Add( int a[] ; int b[] ; int c[] ; int n ; int m)
{
    for (int i=0; i<n; i++)                    n+1        n+1
    for (int j=0; j<m; j++)                    n(n+1)     n2+n
    c[i,j] = a[i,j] +b[i,j];                   n(n)       n2
}
```

FC = 2n2+ 2n +1

T add (…) = $2n^2$+ 2n +1

ASYMPTOTIC NOTATION

1. BIG-OH NOTATION (FORMAL DEFINITION)

- Given functions f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for $n \geq n_0$

Example: The function 8n+5 is O(n).

Justification: By the big-Oh definition, we need to find a real constant c>0 and an integer constant n0 ≥ 1 such that 8n+5 ≤ cn for every integer n ≥ n0. It is easy to see that a possible choice is c = 9 and n0 = 5. Indeed, this is one of infinitely many choices available because there is a trade-off between c and n0.

Example: 10n2 +4n+2 is O(n2).

Justification: Note that 10n2 +4n+2 <= cn^2,

for c = 11, n0 =5 when n ≥ 5.

Example: 2n + 10 is O(n)

2n + 10 ≤ cn

2n + 10 ≤ 3n

For c = 3 and n_0 = 10, n >=10

Example: the function n^2 is not O(n)

n2 ≤ cn

n ≤ c

The above inequality cannot be satisfied since c must be a constant

n^2 is O(n^2).

Example: 7n+2 is O(n)

need c > 0 and n_0 ≥ 1 such that 7n+ 2 ≤ c•n for n ≥ n_0

this is true for c = 8 and n_0 = 2

2- BIG -OMEGA Ω NOTATION (FORMAL DEFINITION)

Given functions f(n) and g(n), we say that f(n) is Ω (g(n)) if there are positive constants c and n_0 such that

f(n)≥ cg(n) for n≥ n_0

Example: The function 8n+5 is Ω (n).

Justification: By the Omega- definition, we need to find a real constant c>0 and an integer constant n0 ≥ 1 such that 8n+5 >= cn for every integer n ≥ n0. It is easy to see that a possible choice is c = 8 and n0 = 1. Indeed, this is one of infinitely many choices available because there is a trade-off between c and n0.

Example: $10n^2 +4n+2$ is Ω (n^2).

Justification: Note that $10n^2 +4n+2 >= cn^2$,

for c = 10, $n_0 = 1$ when n ≥ 5.

Example: 7n+ 2 is Ω (n)

need c > 0 and $n_0 \geq 1$ such that $7n+2 \geq c \bullet n$ for $n \geq n_0$

this is true for c = 7 and $n_0 = 1$

Example: $3n^3 + 20n^2 + 5$ is Ω (n^3)

need c > 0 and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \geq c \bullet n^3$ for $n \geq n_0$

this is true for c = 3 and $n_0 = 1$

3-BIG -THETA Θ NOTATION (FORMAL DEFINITION)

Given functions f(n) and g(n), we say that f(n) is Θ (g(n)) if f (n) is O(g(n)) and f (n) is Ω (g(n)), that is, there are real constants c1 > 0 and c2 > 0, and an integer constant n0 ≥ 1 such that c1g(n) ≤ f (n) ≤ c2g(n), for n ≥ n0.

Example: $10n^2 +4n+2$ is Θ (n^2).

Justification: $10n^2 \leq 10n^2 +4n+2 \leq 11 n^2$ for n ≥ 5.

Example: 8n+5 is $\Theta(n)$.

Justification: $8n \leq 8n+5 \leq 9n$ for $n \geq 5$.

Example: 7n+2 is $\Theta(n)$

need $c > 0$ and $n_0 \geq 1$ such that $c_1 n \leq 7n+2 \leq c_2 n$ for $n \geq n_0$

this is true for c1 = 7 and c2 = 8 and $n_0 = 2$

Common big Ohs

- constant $O(1)$
- logarithmic $O(\log_2 N)$
- linear $O(N)$
- n log n $O(N \log_2 N)$
- quadratic $O(N2)$
- cubic $O(N^3)$
- exponential $O(2^N)$

CASES TO EXAMINE

- **Best case**

  If the algorithm is executed, the fewest number of instructions are executed

- **Average case**

  Executing the algorithm produces path lengths that will on average be the same

- **Worst case**

  Executing the algorithm produces path lengths that are always a maximum

Example: Use Big –O notation to analyze the time efficiency of the following code:

	Inst	Code	F.C.	F.C
1		for (int i=0; i< n ; i++)	n+1	n+1
	2	for int j=0 ; j < n; j++)	n(n+1)	n^2+n
	3	{ s= s+i;	n*n	n^2
	4	p = p + i*j ;	n*n	n^2
		}	3n2+2n+1	

Discarding constant terms produces: $3n^2+2n$

Clearing coefficients: n^2+n

Picking the most significant term: n^2

Big O = $O(n^2)$

COMMON GROWTH RATES