INTRODUCTION TO DATA STRUCTURES

Chapter 01

Prakash Hegade
Minor Degree Program – School of CSE, KLE Tech.

Pointers

What is a Variable?

Syntax of a variable can be given as:
data_type variable_name = value;

Example:

int number = 100;
To print the value present in variable 'number' we can use,
printf("%d", number);

Every variable is associated with a value and has an address. Whenever we refer a variable, it means we are referring the value of that variable but not its address.

How to get address of the variable?
int number = 100;
&number gives address of the variable.

We can store this address in another variable say **'p'.** i.e. p = &number;
During compilation memory for 'number' and 'p' are allocated. 'p' contains address of variable 'number'. Such variables are called as pointer variables.

Definition: "A pointer is a variable which contains address of another variable"

Declaring and initializing:

How to distinguish a variable and a pointer variable??
A pointer variable is always prefixed with a '*'.
Syntax: *data_type * pointer_name;*
int number= 100;
int * p;
p = &number;

Note:

- int * p; *// p is a pointer variable which can hold the address of a variable of type int*
- double * q; *//q is a pointer variable which can hold the address of a variable of type double*
- Following are same:
  int *pa; int * pa; int * pa;
- When pointers are declared, it is better and safer to initialize it with NULL
- Global pointers are initialized to NULL during compilation

Program: Accessing values using pointers

```
#include <stdio.h>
int main()
{
  int value, number=100;
  int * p;
  p = &number;
  value = *p;

  //100 can be obtained by all below statements:
  printf("%d\n", number);
  printf("%d\n", *p);
  printf("%d\n", value);
  printf("%d\n",*&number);
}
```

Address Arithmetic

Following operations are valid on Pointers:

- A pointer can be incremented or decremented
- Addition of integers to pointers is allowed
- Subtraction of two pointers of same data type is allowed
- The pointers can be compared using relational operators
    - p>q, p==q , p<=q, p<q, p==NULL, p!=q (p and q are two pointers)

Following operations are invalid on Pointers:

- Arithmetic operations such as addition, multiplication and division on two pointers are not allowed
- A pointer variable cannot be multiplied / divided by a constant or a variable
- Address of the variable cannot be altered. Example: &p = 2048

Pointers and Arrays

Compiler allocates memory to an array contiguously. The address of the array can be obtained by &a[0], &a[1] etc or also by specifying a, a+1 etc.

So, we can access address of ith element either by (a+i) or &a[i]

Note:

- Base address of array will be stored in the name given to the array
- Array name “**a**” can be considered as a pointer
- Array name “**a**” is pointer to the first element in the array
- Array and pointers go hand in hand

On base address “**a**”,
a++ is not valid
a = &a[5] is not valid or any operation which modifies the address is not valid because we cannot change the base address. It is a **constant pointer**.

Program: Program to Print Array Addresses
```
#include <stdio.h>
int main()
{
   int array[4];
   int index;
   printf("The base address is %p\n", array);
   printf("All the array member addresses\n");
   for(index =0; index<4;index++) {
      printf("%p\n", array+index);
   }
   return 0;
}
```

Sample Output:
```
The base address is 0022FF0C
All the array member addresses
0022FF0C
0022FF10
0022FF14
0022FF18
```

Generalizing:
(array + index) = base address + index * number of bytes required to store an element

Note:
```
int array[4] = {1,2,3,4};
int *p;
p = array;        // p = &array[0];
```
Now the operations, p++, p = &a[2] are valid.

Program: Printing the array elements using pointers
```
#include <stdio.h>
int main()
{
   int array[] = {1,2,3,4};
   int index;
   int *p;
   p = array;

   // Method 01
   for(index =0; index<4;index++) {
      printf("%d\t", *p);
      p++;
```

```
    }

    // Method 02
    printf("\n");
    p = p - 4;
    for(index =0; index<4;index++) {
        printf("%d\t", *(p+index));
    }

    // Method 03
    printf("\n");
    for(index =0; index<4;index++) {
        printf("%d\t", index[p]);
    }

    // other valid statements: p[index], * (index+p)
    return 0;
}
```

Strings

- Array of characters
- Each String ends with a null character (\0) – only way for the string functions to know where the string will end.
- If no '\0' it is not a string, it is collection of characters
- Defined in header <string.h>

Declaring and initializing:

```
char course[5] = {'D', 'S', '\0'};      OR
char course[5] = {"DS"};                OR
char course[5]= "DS";
```

Program: Program to print a string using pointers

```
#include <stdio.h>
int main()
{
    char course[] = "DS";
    char *p;
    p = course;
    while(*p != '\0') {
        printf("%c", *p);
        p++;
    }
```

```
   return 0;
}
```

All below are also valid:
course[i], i[course]
* (course + i), *(i + course)

String I/O functions:

To read:

- scanf( ) with %s format specification
- gets( )
- getchar( )

To write:

- printf( ) with %s format specification
- puts( )
- putchar( )

Note: To accept strings with space we generally use: gets(string); But using gets() is a bad programming practice.
Because it has the buffer limitation and can overwrite data even without any warnings. So a version of scanf() as specified below can be used to accept strings with space in it.
scanf("%[^\n]s", name);

Array of Strings:

Example:
char pens[3][20] = {“Reynolds”, “Parker”,”Cello” };
These strings can be accessed using only first subscript Example: printf(“%s”, pens[1]); will output Parker

String Manipulation Functions

Program: implementing the strlen(str) - get the length of the string

```
#include <stdio.h>
int main()
{
   char str[30];
   int counter = 0;
   char *p = str;

   printf("Enter the string\n");
   scanf("%[^\n]s", str);

  while(*p != '\0')
```

```
 {
     counter++;
     p++;
  }

 printf("Length of the string is %d\n", counter);
 return 0;
}
```

Program: implementing strcpy (str2, str1) - copies str1 to str2

```
#include <stdio.h>
int main()
{
   char str1[30];
   char str2[30];
   char *ptr1;
   char *ptr2;

   ptr1 = str1;
   ptr2 = str2;

   printf("Enter the String 1\n");
   scanf("%s", str1);

   while(*ptr1 != '\0')
   {
      *ptr2 = *ptr1;
      ptr1++;
      ptr2++;
   }
   *ptr2 = '\0';
   printf("The copied String is.. %s\n", str2);
   return 0;
}
```

Program: implementing strcat(str1, str2) - append str2 to str1

```
#include <stdio.h>
int main()
{
   char str1[40];
   char str2[20];

   char *ptr1;
   char *ptr2;
```

```
  ptr1 = str1;
  ptr2 = str2;

  printf("Enter the first string\n");
  scanf("%s", str1);

  printf("Enter the second string\n");
  scanf("%s", str2);

 // Reach to the end of the str1
  while(*ptr1 != '\0')
     ptr1++;

  // Append the second string to first
  while(*ptr2 != '\0') {
     *ptr1 = *ptr2;
     ptr1++;
     ptr2++;
  }
  *ptr1 = '\0';

  printf("The concatenated string is... %s", str1);
  return 0;
}
```

Other Functions which you must know:

- **strncmp**: compares first n characters of two string inputs
  - Syntax: strncmp(str1, str2, n)
  - Returns : -1, 0 or 1
- **strncpy:** copies first n characters of the second string to the first string
  - Syntax: strncpy(str1, str2, n)
- **strncat:** appends first n characters of the second string at the end of first string.
  - Syntax: strncat(str1, str2, n)
- **strlwr(str) :** convert any uppercase letters in the string to the lowercase
- **strupr(str) :** convert any lowercase letters that appear in the input string to uppercase
- **strchr(str, char):** searches for specified character in the string. It returns NULL if the desired character is not found in the string.

Pointer Arrays

Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

Can you differentiate between the following?

int a;   int a[10];   int *a;   int *a[10];

Program: Usage of array of pointers

```
int main() {
        int *arr[3];
        int i = 10, j = 20, k = 30;
        int index;
        arr[0] = &i;
        arr[1] = &j;
        arr[2] = &k;
        for(index = 0; index < 3; index++)
                printf("%d\n", *(arr[index]));
        return 0;
}
```

Operations Comparison Table

	int array[i][j];	**int *array[i]**
	Array Indexing	Pointer Arithmetic
Address	&array[i][j]	array[i] + j
Value	array[i][j]	*(array[i]+ j)

Example of strings:

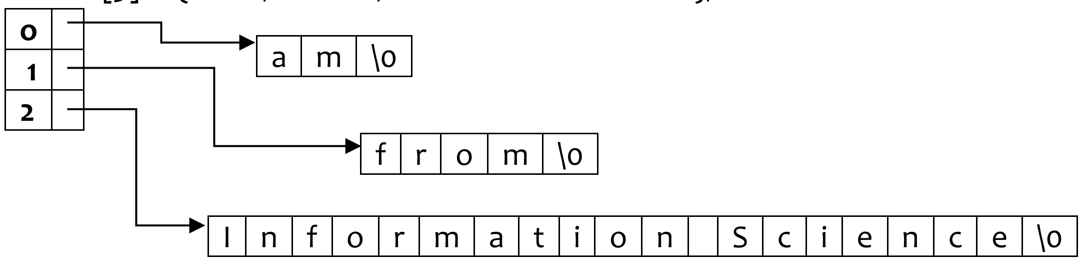
Similar concept can be extended to strings also.

Consider for example,
A 2D- char array,
char a[3][20] = {"am", "from", "Information Science"};

The size of each row will be fixed during compilation and results in wastage of memory.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
0	a	m	\0																	
1	f	r	o	m	\0															
2	I	n	f	o	r	m	a	t	i	o	n		S	c	i	e	n	c	e	\0

To avoid the wastage of memory, we can go for,
char *a[3] = {"am", "from", "Information Science"};



Now **a** is an array of character pointers. Each location in this array points to an array of items of a specific data type.

Structures

The detailed notes on structures can be downloaded from the ebook 'Structures and C' found here:
https://www.smashwords.com/books/view/644937

Algorithms: Introduction

Algorithm is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in a finite amount of time.

Al-khwarizmi defines it with the following components:

- Problem
- Unambiguous instructions
- Legitimate input
- Finite time
- Output

Every algorithm is associated with:

- Properties like simplicity, correctness, generality, concise, clear, unambiguous etc
- Representation mechanism
- Design techniques involved
- Capability of computational device
- Picking right data structure
- Exact/ approximate solutions
- Analysis

Algorithm is associated with two types of efficiencies, time and space.
Time Efficiency
How fast an algorithm runs

Space efficiency
Extra space the algorithm requires

Step Count and Basic Count:

Step count refers to time taken for every operation execution in the code.

Basic count is time for most time consuming or critical operation in the code. We generally use Basic count to refer to time efficiency of a code.

Fundamentals of Algorithmic Problem Solving

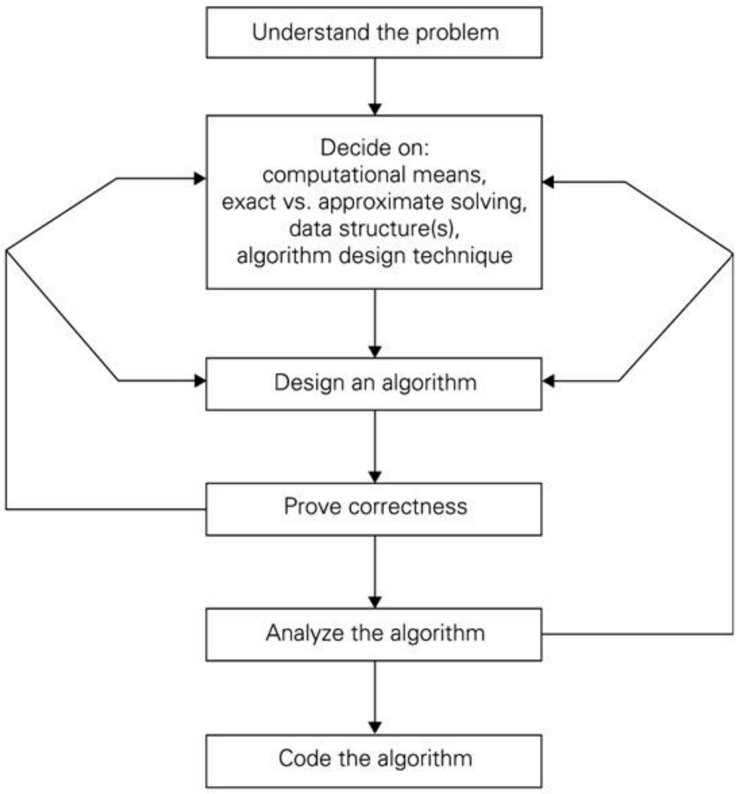

Understand the problem
Decide on: computational means, exact vs. approximate solving, data structure(s), algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

Fig. 1: Referenced from "Introduction to Design and Analysis of algorithms" by Anany Levitin

Orders of Growth

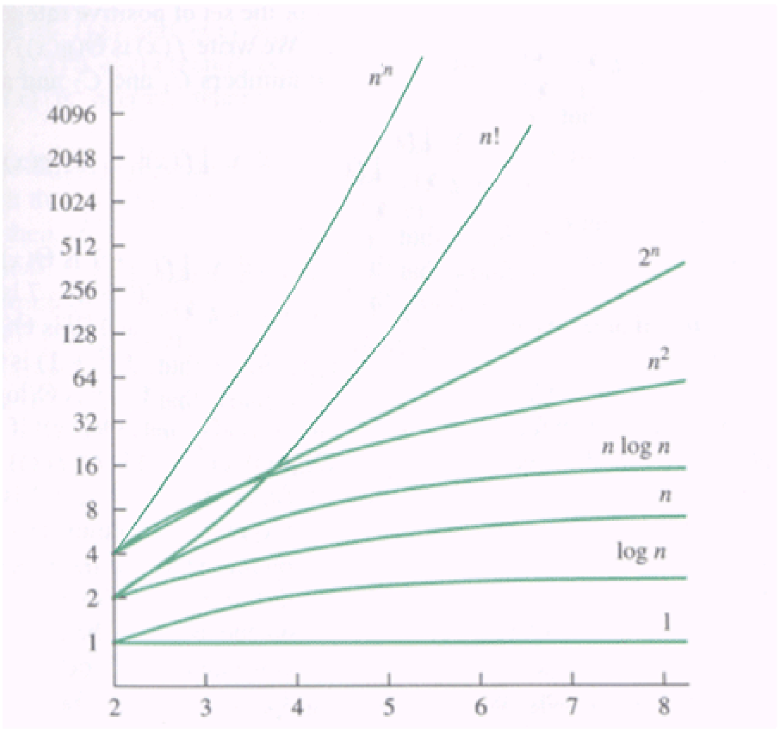


Fig 2: Orders of Growth

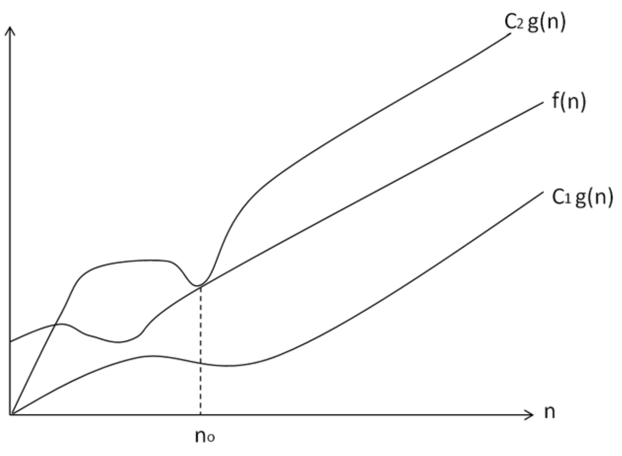
Asymptotic Notations

Let's start with the basic question, 'What are Asymptotic Notations?'
Asymptotic notations are mathematical notations that describe the limiting behaviour of a function when the argument tends towards a particular value (or may be infinity). The phrase 'limiting behaviour' here is important. The notations describe the restriction and boundaries towards a function.

Let's stay for a while in the math world and understand the three members that belong to this family.

Member 01: θ Notation

A function f(n) belongs to the set of θ(g(n)) if there exists positive constants C1 and C2 such that it can be packed between two functions **C1g(n)** and **C2g(n)**, this being true from the value n_0. The notation is graphically presented in the fig. 3 given below.



f(n) = θ (g(n))

Fig. 3: Θ notation

For all n >= n_0, the value of f(n) lies at or above C1g(n) and at or below C2g(n). i.e,
C1 g(n) <= f(n) <= C2 g(n) for all n >= n_0

Example:

As an example let us express the function f(n) = 10n + 10 using θ notation.
10n <= 10n+10 <= 15n
C1 = 10
C2 = 15
g(n) = n
n_0 = 1

Member 02: O Notation

O notation is used to give an upper bound on a function, to within a constant factor. For all the values of n at and to the right of n_0, the value of the function f(n) is on or below Cg(n). The notation is graphically presented in the fig. 4 given below.

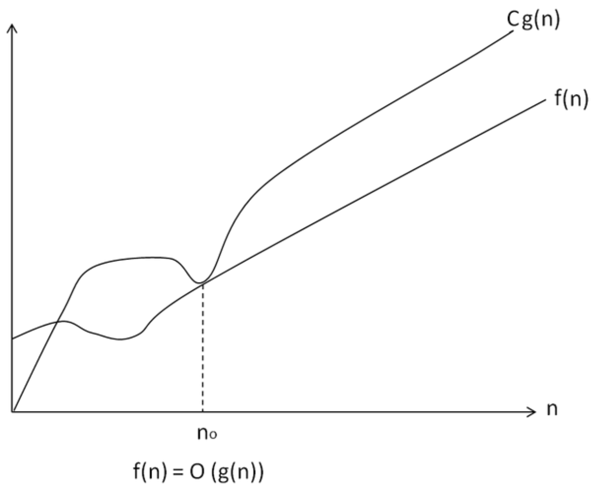


Fig. 4: O notation

For all n >= n_0, the value of f(n) lies at or below Cg(n) i.e,
f(n) <= C g(n) for all n >= n_0

Example:
As an example let us express the function f(n) = 10n + 10 using O notation
10n+10 <= 15n
C = 15
g(n) = n
n_0 = 1

Member 03: Ω Notation

Ω notation is used to give a lower bound on a function, to within a constant factor. For all the values of n at and to the right of n_0, the value of the function f(n) is on or above Cg(n). The notation is graphically presented in the fig. 5 given below.

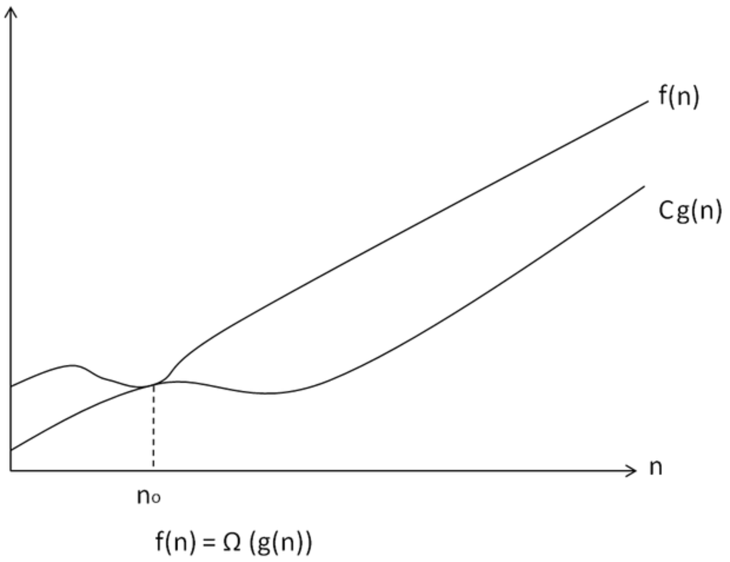


Fig. 5: Ω notation

For all n >= n_0, the value of f(n) lies at or above Cg(n). i.e,
f(n) >= C g(n) for all n >= n_0

Example:
As an example let us express the function f(n) = 10n + 10 using Ω notation.
$10n+10 >= 10$
$C = 10$
$g(n) = n$
$n_0 = 1$

These notations that we understood from the math world can be used to characterize aspects or features of any domains applicable and relevant.

Asymptotic Notions for Algorithms
In algorithms we use asymptotic notations to characterize the running times of the algorithms. However, that said, they can be used to characterize some other or any other feature like space as well.
When we say running time, we need to be little more specific about the nature of the input – Is it the worst case, best case or an average case. Considering the cases and the notations, we can easily map the three notations to the three cases.

Hence we have the notations as:

Θ – Average Case
O – Worst Case
Ω – Best Case

Algorithmic Conventions and Analysis

Consider an algorithm to find the maximum element in an array.

```
ALGORITHM MaxElement(A[0… n-1])
// Determines the value of largest element in a given array
// Input: An array A[0…n-1] of real numbers
// Output: The value of largest element in A

maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Analysis:

1. Find out the **basic operation**. There is no hard and fast rule or any definition to determine the BO. But it is the one which consumes maximum number of clock cycles (or say takes longest time or runs many times while code executes)
   In above case, it is the comparison operation**:** ***if A[i] > maxval***
2. Secondly, we need to check do we have separate best, worst and average case analysis?
   The algorithm is not going to consume different time at each run for same input size ‘n’. Hence we have only one analysis to be made.

3. Let C(n) be the function for number of times the comparison is made. Setting up the equation:
   The BO is going to be executed once in every iteration of for() which runs for the condition variable 'i' ranging from 1 to n-1. Hence we can set up a summation formula.

   C(n) = $\sum_{i=1}^{n-1} 1$
   = (n-1 -1 +1) *[Upper Bound – Lower Bound + constant]*
   = n -1

4. The order of growth is linear in nature. As the input size increases, the time taken also increases in linear fashion.

Example 02:
ALGORITHM Binary(n)
// Counts the number of digits in binary representation of a given positive decimal integer
// Input: a positive decimal integer
// Output: Number of digits in a binary representation of a given positive decimal integer

```
if n=1
   return 1
else
  return Binary (n/2) + 1
```

[**Note:**
The iterative version of above algorithm would be written as:

```
count ← 1
while n > 1
   count ← count + 1
   n ← n / 2
return count ]
```

Analysis:
Here Addition operation is the Basic operation. Setting up the recurrence relation:

$$B(n) = \begin{cases} 0 & \text{if } n = 1 \\ B(n/2) + 1 & \text{otherwise} \end{cases}$$

(How did we set up the equation?
when n = 0, there are no additions hence it is 0.
In all other cases there is one addition and recursive function call by halving the input size.)

Solving,

$B(n) = B(n/2) + 1$

Assume $n = 2^k$

$B(2^k) = B(2^{k-1}) + 1$

The further solving steps can be referred from class examples.
We finally end up with order of growth of **$\log_2 n$.**

The Towers of Hanoi Problem

Task:

There are 3 pegs and N disks. The larger disk is always at the bottom. We need to move the N disks from A to C using B as auxiliary with the constraint that the smaller disk is always at the top.

Solution:

1. If n == 1, move the single disk from A to C and stop
2. Move the top n-1 disks from A to B using C as auxiliary
3. Move the remaining disk from A to C
4. Move the n-1 disks from B to C using A as auxiliary

(If you are not able to understand above procedure, apply the solution on 3 disks problem. The procedure will follow.)

Designing a program:

- Design a better presentation of the solution
- Think on what will be the input and output to the program
- Present the output in user understandable way
- The proper input / output format may make the program design much simpler

Program:

```
#include <stdio.h>
#include <stdlib.h>
void towers(int, char, char, char);

int main()
{
   int n;
   printf("Enter the number of Disks to be moved\n");
   scanf("%d", &n);
   towers(n, 'A', 'C', 'B');
   return 0;
}
```

```
void towers(int n, char from, char to, char aux)
{
  if( n == 1)
  {
    printf("Move disk 1 from %c to %c\n",from, to);
    return;
  }

  // Move top n-1 disks from A to B using C as auxiliary
  towers(n-1, from, aux, to);

  // Move remaining disk from A to C
  printf("Move disk %d from %c to %c\n", n, from, to);

  // Move n-1 disks from B to C using A as auxiliary
  towers(n-1, aux, to, from);
}
```

Efficiency Analysis:

Here the basic operation is the moment of the disk.

Setting up the recurrence relation:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) + 1 + T(n-1) & \text{otherwise} \end{cases}$$

$T(n) = 2\,T(n-1) + 1 \quad \rightarrow (1)$

Substitute n = n-1 in (1), we have

$T(n-1) = 2T(n-2) + 1 \quad \rightarrow (2)$

Substitute (2) in (1) we have,

$$\begin{aligned} T(n) &= 2\,[\,2\,T(n-2) + 2] + 1 \\ &= 2^2\,T(n-2) + 2 + 1 \\ &= 2^3\,T(n-3) + 2^2 + 2 + 1 \\ &= 2^4\,T(n-4) + 2^3 + 2^2 + 2 + 1 \\ &\ldots \\ &\ldots \\ &= 2^n\,T(n-n) + 2^{n-1} + 2^{n-2} + \ldots. + 2^3 + 2^2 + 2 + 1 \\ &= 2^n\,T(0) + 2^{n-1} + 2^{n-2} + \ldots. + 2^3 + 2^2 + 2 + 1 \\ &= (2^n * 0) + 2^{n-1} + 2^{n-2} + \ldots. + 2^3 + 2^2 + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \ldots. + 2^3 + 2^2 + 2 + 1 \end{aligned}$$

This is a GP series.

$S = a\,(r^n - 1) / r - 1$

Where a = 1, r = 2 and n = number of terms.
Solving:
S = 1 $(2^n - 1) / 2-1$
 $= 2^n - 1$
Algorithm has order of growth which is exponential in nature.

Exercise:

Perform Mathematical Analysis for following algorithms: (Refer class notes for solutions)

Example 01:

```
ALGORITHM UniqueElements (A[0… n-1])
// Determines if all the elements in array are distinct
// Input: An array A[0… n-1] of real numbers
// Output: returns true if all array elements are distinct, false otherwise
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
    if A[i] = A[j]
        return false
return true
```

Example 02:

```
ALGORITHM MatrixMultiplication (A[0… n-1, 0… n-1], b[0… n-1, 0… n-1]))
// Multiplies two n*n matrices
// Input: Two n-by-n matrices A and B
// Output: Matrix C = AB
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i, j] ← 0.0
         for k ← 0 to n-1 do
            C[i, j] ← C[i, j] + A[i, k]  * B[k, j]
return maxval
```

Example 03:

```
ALGORITHM Factorial (n)
// Computes factorial of the given number
// Input: positive integer n
// Output: factorial of supplied n
If n = 0
    return 1
return n * factorial(n-1)
```

~*~*~*~*~*~*~