DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I

Chapter-1 Introduction to Algorithms

Dr B.Kumar Babu

Assistant Professor,

Dept. of CSE,

School Of Technology,

GITAM(Deemed to be) University,

Hyderabad.

Agenda

- Algorithm specification
- Performance Analysis
- **Divide and Conquer:**
  - The general method
  - Binary search,
  - finding maximum and minimum,
  - Merge sort,
  - Quick sort,
  - Selection sort,
  - Strassen's Matrix multiplication.

Basic Data Structures concepts

1. *Linear and Non-linear data structures*: In linear data structures the data elements are arranged in a linear sequence like in an array, data processed one by one sequentially. Linear data structures contain following types of data structures:
   - Arrays
   - Linked lists
   - Stacks
   - Queues

   In non-linear data structures, the data elements are not in sequence that means insertion and deletion are not possible in a linear manner. Non-linear data structures contain following types of data structures:
   - Tree
   - Graph
2. *Homogenous and Non-homogenous data structures*: In homogenous data structures the data elements are of same type like an array. In non-homogenous data structures, the data elements may not be of same type like structure in 'C'.
3. *Static and Dynamic data structures*: Static structures are ones whose sizes and structures associated memory location are fixed at compile time e.g. array. Dynamic structures are ones that expand or shrink as required during the program execution and their associated memory location change, e.g. linked list.

Data Structure Operations

Algorithms manipulate the data that is represented by different data structures using various operations. The following data structure operations play major role in the processing of data:

- *Creating*. This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data elements.
- *Inserting*. Adding a new data element to the data structure.
- *Updating*. It changes data values of the data structure.
- *Deleting*. Remove a data element from the data structure.
- *Traversing*. The most frequently used operation associated with data structure is to access data elements within a data structure. This form of operation is called traversing the data structure or visiting these elements once.
- *Searching*. To find the location of the data element with a given value, find the locations of all elements which satisfy one or more conditions in the data structure.
- *Sorting*. Arranging the data elements in some logical order e.g. in ascending or descending order of students' name.

What is an Algorithm?

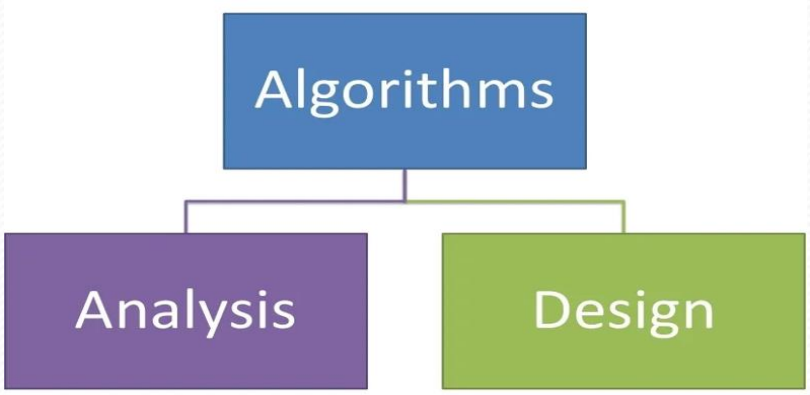
Definition

An algorithm is a finite set of instructions that takes some raw data as input and transforms it into refined data. An algorithm is a tool for solving a well-specified computational problem. Every algorithm must satisfy the following criteria:

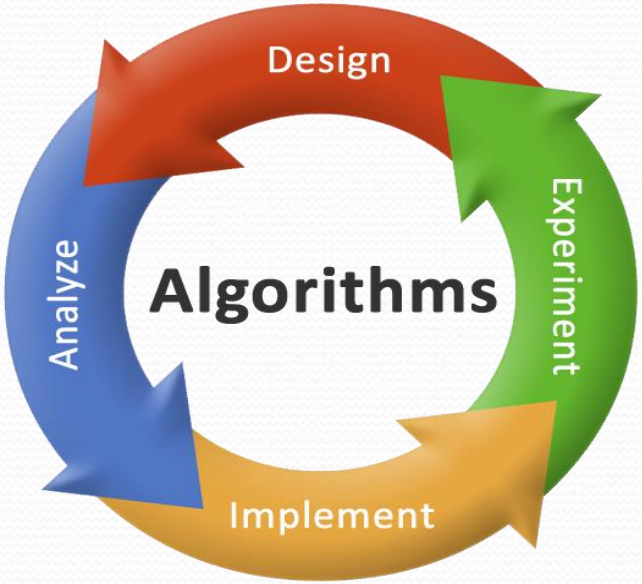
- **Input:** In every algorithm, there must be zero or more data that are externally supplied.
- **Output:** At least one data is produced.
- **Definiteness:** Each instruction must be clear and unambiguous (i.e., clearly one meaning).
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.
- **Effectiveness:** Every instruction must be feasible and provides some basis towards the solution.

(OR)

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way.



- **Analysis:** predict the cost of an algorithm in terms of resources and performance
- **Design:** creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.



Algorithm vs Pseudocode

- An algorithm is a formal definition with some specific characteristics that describes a process. Generally, the word "algorithm" can be used to describe any high level task in computer science.

Algorithm: Selection Sort

Input: A list L of integers of length n

Output: A sorted list L1 containing those integers present in L

Step1: Find the minimum value in the list L

Step2: Swap it with the value in the current position

Step3: Repeat this process for all the elements until the entire list is sorted

Step 4: Return the sorted list L1

Step 5: Stop

- Pseudocode is an informal and human readable description of an algorithm leaving many details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm.

Pseudocode : Selection Sort

```
for j ← 1 to n-1
    smallest ← j
        for i ← j + 1 to n
            if A[ i ] < A[ smallest ]
                smallest ← i
        Exchange A[ j ] ↔ A[ smallest ]
```

DAA using Programming languages

- All algorithms can be coded using any programming language such as
  - C,
  - Python,
  - C++,
  - Java,
  - PHP,
  - JavaScript,
  - C#,
  - Etc.....

Algorithm

- Definition

  An *algorithm* is a finite set of instructions that accomplishes a particular task.

All algorithms must satisfy the following criteria:

1. Input : Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness : Each instruction is clear and unambiguous.

Statements such as **"add 6 or 7 to x"** or **"Compute 5/0"** are not permitted"

4. Finiteness : The algorithm should terminate after a finite number of steps.

5. Effectiveness: Instruction is basic enough to be carried out.

Pseudo code for expressing algorithms

- We present most of our algorithms using pseudo code that resembles C and Pascal.

1. Comments begin with // and continue until end of the line.
2. Blocks are indicated with matching braces: { and }.
   i. A compound statement
   ii. Body of a procedure.

3.

i. An identifier begins with a letter.
ii. The data types of variables are not explicitly declared.
iii. The types will be clear from the context.
iv. Whether a variable is global or local to a procedure will also be evident from the context.
v. We assume simple data types such as integer, float, char, boolean, and so on.
vi. Compound data types can be formed with records.

```
node = record
        {
                datatype_1 data_1;
                        :
                datatype_n data_n;
                node      *link
        }
```

data items of a record can be accessed with → and period(.)

C style :-

```
struct  node
{
    datatype_1 data_1;
            :
    datatype_n  data_n;

    struct node  *link

}
```

4. Assignment of values to variables is done using the assignment statement.

< variable > := < expression >

5. There are two boolean values **true** and **false.** To produce these values, logical operators **and**, **or** and **not** and the relational operators <, ≤,=, ≠, ≥ and > are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example the (i,j)th element of the array A is denoted as A[i,j].

7. The following looping statements are used: for, while, and repeat until.

The general form of a **while** loop:

```
while( condition ) do
{
        statement_1;
            :
        statement_n;
}
```

The general form of a **for** loop:

for *variable* := *value1* **to** *value2* **step** *step* **do**

- Here value1, value2, and step are arithmetic expressions.
- The clause "**step** *step*" is optional and taken as +1 if it does not occur.
- *step* could be either positive or negative.
- *variable* is tested for termination at the start of each iteration.

The for loop can be implemented as a while loop as follows:

```
variable:=value1;
incr:=step;
while( ( variable – value2)*step ≤ 0) do
{
        <statement 1>
                    :
    <statement n>
    variable :=variable+incr;
}
```

- The general form of a **repeat-until** loop:

```
repeat
        <statement 1>
              :
        <statement n>
until ( condition )
```

The statements are executed as long as condition is false.

8. A conditional statement has the following forms:

- **if** < *condition* > **then** < *statement* >

- **if** < *condition* > **then**
  < *statement* 1>
  else
  < *statement* 2>

9. Input and output are done using the instructions **read** and **write.**

Ex: read n;

write n;

10. Procedure or function starts with the word **Algorithm.**

General form :

Algorithm *Name*(<parameter list>)

{

body

}

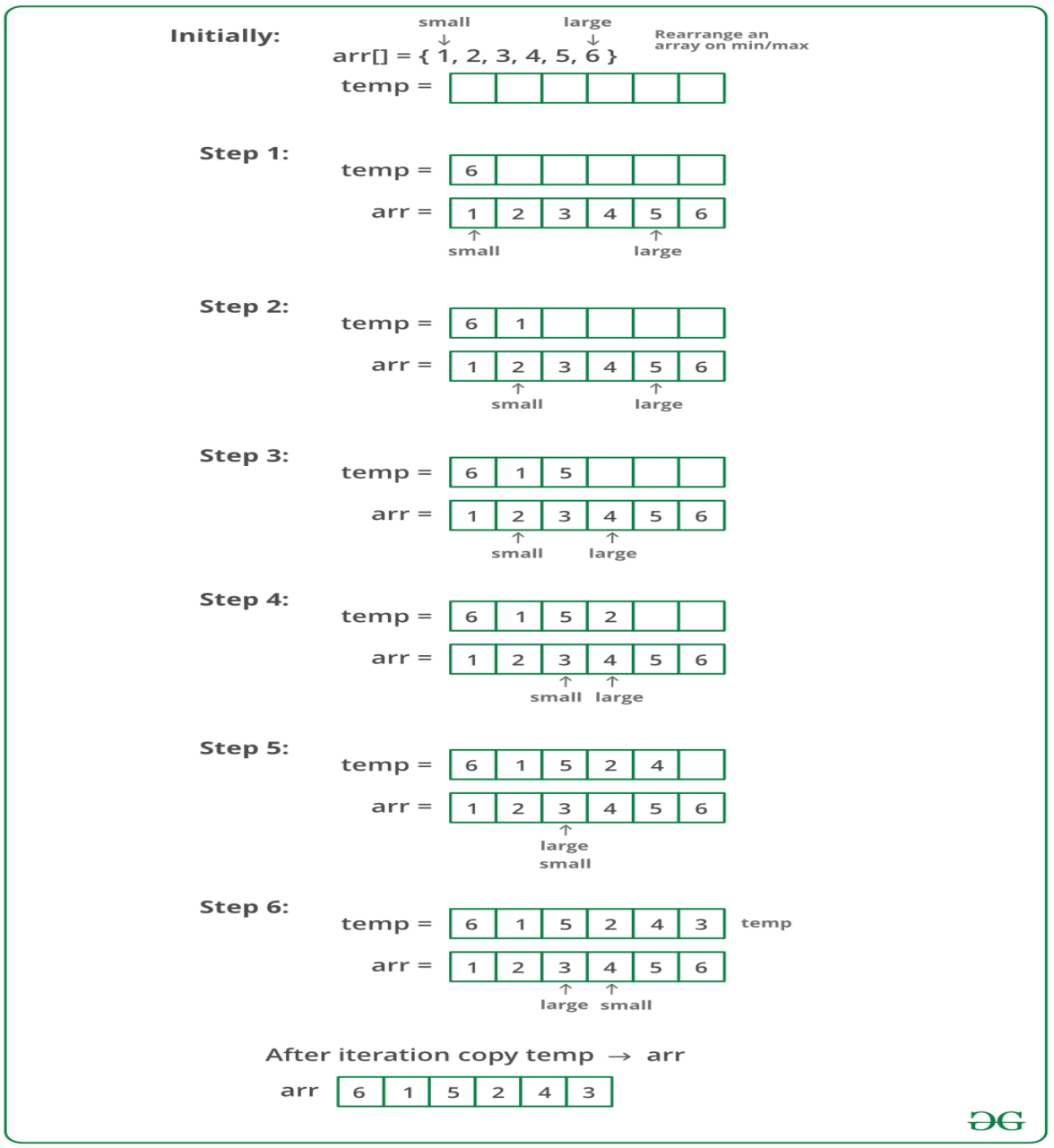
where *Name* is the name of the procedure.

- Simple variables to procedures are passed by value.
- Arrays and records are passed by reference

Ex:-Algorithm that finds and returns the maximum of n given numbers

```
Algorithm max(a,n)
// a is an array of size n
{
        Result:=a[1];
        for i:=2 to n do
            if a[i]>Result  then Result:=a[i];
       return Result;
}
```

- Process for above example



Ex:-Write an algorithm to sort an array of n integers using bubble sort.

Algorithm BubbleSort(a,n)

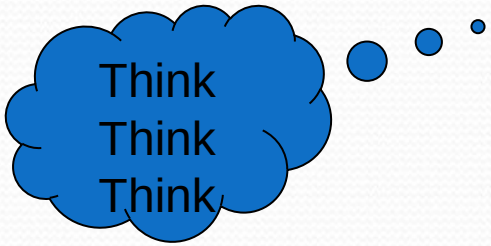
```
// a is an array of size n
{
        for i:=1 to n-1 do
        {
                for j:=1 to n-i do
                {
                    if( a[j] >a[j+1] ) then

                            t=:a[j];
                        a[j]:=a[j+1];
                        a[j]:=t;
                }
        }

}
```

Performance Analysis (machine independent)

- There are many things upon which the performance will depend.
  - Does the program efficiently use ***primary and Secondary*** storage?
  - Is the program's ***running Time*** acceptable for the task?
  - Does it do ***what*** we want it to do?
  - Does it ***work correctly according to the specifications*** of the task?
  - Does the program contain ***documentation*** that shows ***how to use it*** and ***how it works***?
  - Is the program's code ***readable***?

How to achieve them?

- Good programming style, experience, and practice
- Discuss and think.



- There are other criteria for judging algorithms that have a more direct relationship to performance.
  - 1. Space Complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion. Therefore, the space requirement of any problem P can be written as

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

  - 2. Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion. Therefore, the time requirement of any problem P can be written as

$$T(P) = C + T_P(\text{Instance characteristics(I)})$$

Space Complexity

- The space needed by a program has the following components.
  - Instruction space
    - Instruction space is the space needed to store the compiled version of the program instructions.
    - The amount of instruction space that is needed depends on the compiler used to compile the program into machine code.

Data space

- Data space is the space needed to store all constant and variable values. Data space has two components.
  - Space needed by constants(ex; 1 and 2 in max of n num algorithm) and simple variables(such as i, j, n etc).
  - Space needed by a dynamically allocated objects such as arrays and class instances.
- Space taken by the variables and constants varies from language to language and platform to platform.

- Environmental stack space

  - The environment stack is used to save information needed to resume execution of partially completed functions.
  - Each time a function is invoked the following data are saved on the environment stack.
    - The return address .
    - The values of all local variables and formal parameters in the function being invoked(necessary for recursive functions only).

For example Fact using Recursive algorithm

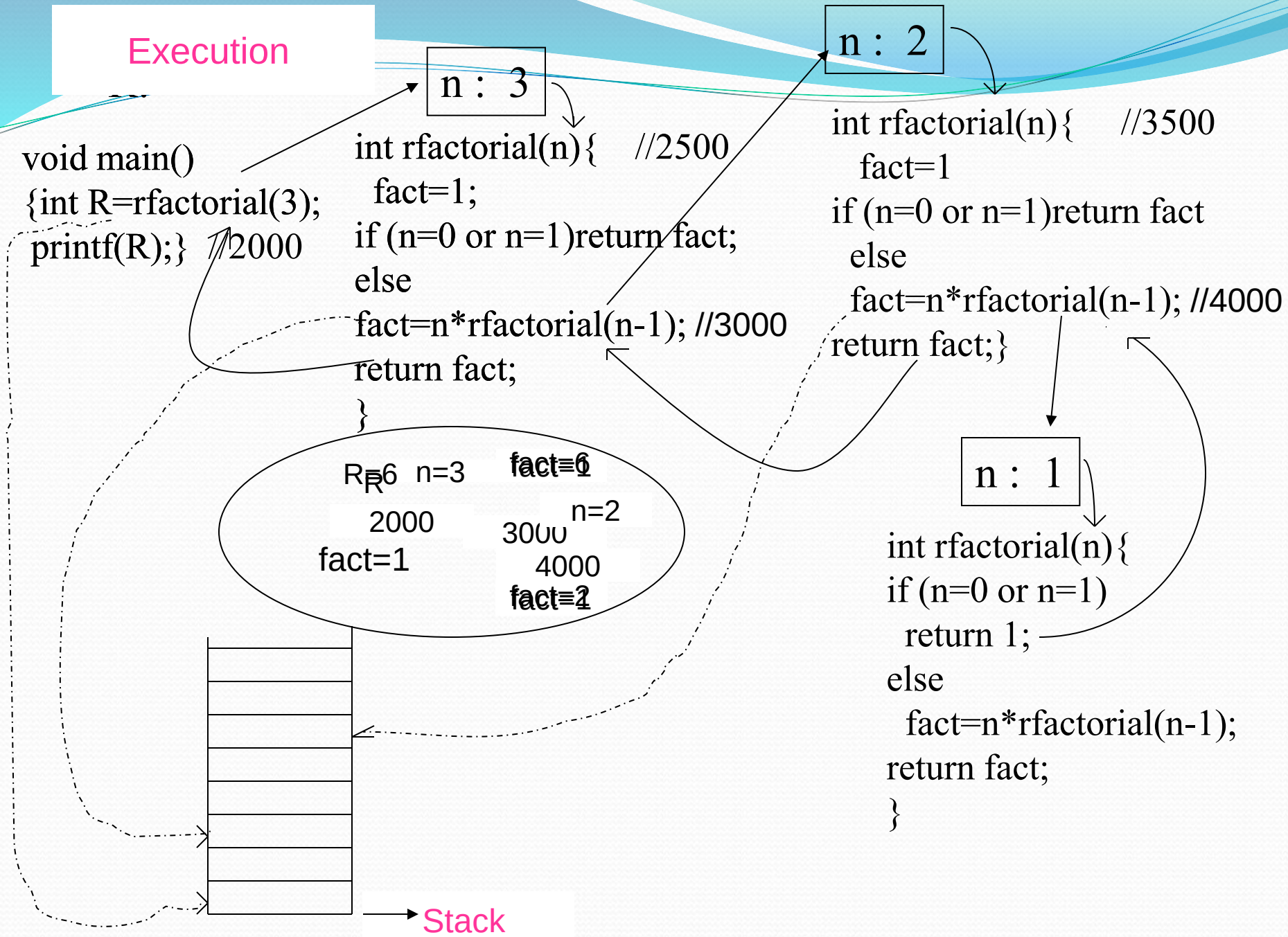
Algorithm rfactorial(n)

```
// n is an integer
{          fact=1;
          if (n=1 or n=0) return fact;
                    else
                    fact=n*rfactorial(n-1);
          return fact;
}
```

Each time the recursive function rfactorial is invoked,

the current values of n and fact and the program location to return to on completion are saved in the environment stack.

Execution


n : 3
n : 2
n : 1
void main()
{int R=rfactorial(3);
printf(R);} //2000
int rfactorial(n){ //2500
fact=1;
if (n=0 or n=1)return fact;
else
fact=n*rfactorial(n-1); //3000
return fact;
}
int rfactorial(n){ //3500
fact=1
if (n=0 or n=1)return fact
else
fact=n*rfactorial(n-1); //4000
return fact;}
int rfactorial(n){
if (n=0 or n=1)
return 1;
else
fact=n*rfactorial(n-1);
return fact;
}
R=6 n=3 fact=1
2000
fact=1
3000
n=2
4000
fact=2
Stack

Summary of space complexity

- The space needed by a program depends on several factors.
- We cannot make an accurate analysis of the space requirements of a program unless we know the computer or compiler that will be used.
- *However, we can determine the components that depend on the characteristics of the* problem instance **(e.x.the number of inputs and outputs or magnitude of the numbers involved)** *to be solved.*

Ex:-1. Space requirements of a program that sorts n elements can be expressed as a function of n.

2. Space requirements of a program that adds two m×n matrices can be expressed as a function of m, n.

- The size of the instruction space is independent of the problem instance to be solved.
- The contribution of the constants and simple variables to the data space is also independent of the problem instance to be solved.
- Most of the dynamically allocated memory(ex., arrays, class instances etc) depends on problem instance to be solved.
- The environmental stack space is generally independent of the problem instance unless recursive functions are in use.

Therefore, We can divide the total space needed by a program into two parts:

i) Fixed Space Requirements (C)

Independent of the characteristics (num, size of inputs and outputs) of the problem instance (I)

- instruction space
- space for simple variables and constants.

ii) Variable Space Requirements ($S_P(I)$)

depend on the characteristics of the problem instance (I)

- Number of inputs and outputs associated with I
- recursive stack space (formal parameters, local variables, return address).

- Therefore, the space requirement of any problem P can be written as

$$S(p)=C+S_p(\text{Instance characteristics})$$

- Note:
  - When analyzing the space complexity of an algorithm, we concentrate only on estimating

    S_p (Instance characteristics).
  - We do not concentrate on estimating fixed part c .
  - We need to identify the instance characteristics of the problem to measure S_p

Example1

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

- Problem instance characterized by the specific values of a,b,and c.
- If we assume one word (4 bytes) is adequate to store the values of each a, b, and c , then the space needed by abc is independent of the instance characteristics.

Therefore, $S_{abc}(\text{ instance characteristics})=0$

Example2

```
Algorithm sum(a,n)
{
    s:=0;
    for i:=1 to n do
            s:=s+a[i];
      return s;
}
```

- Problem instance characterized by n.
- The amount of space needed by n is one word, since n is integer.
- The space needed by a is the space needed by the variables of type array of floating point numbers.
- It is atleast n words, since a must be large enough to hold the n elements to be summed.
- Therefore, $S_{sum}(n) = n+3$

Example3

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

Type	Name	Number of bytes
formal parameter: int	a	2
formal parameter: int	n	2
return address (used internally)		2
Total per one recursive call		6

Total no.of recursive calls n, therefore $S_{RSum}(n)=6(n+1)$

Time Complexity

$$T(P)=C+T_P(I)$$

- The time, $T(P)$, taken by a program, P, is the sum of its compile time C and its run (or execution) time, $T_P(I)$.
- The compile time does not depend on the instance characteristics.
- We will concentrate on estimating run time $T_p(I)$.

- If we know the characteristics of the compiler to be used, we can determine the
  - No. of additions , subtractions, multiplications, divisions, compares, and so on.
- Then we can obtain an expression for $T_p(n)$ Of the form

  $T_P(n)=c_aADD(n)+ c_sSUB(n)+ c_mMUL(n)+ c_dDIV(n)+\ldots\ldots$

  where,
  - n denotes the instance characteristics.
  - c_a, c_s, c_m, c_d and so on denote the time needed for an addition , subtraction, multiplication, division and so on. And
  - ADD, SUB, MUL, DIV, and so on are functions whose values are the no.of additions, subtractions, multiplications, divisions, and so on.

- Obtaining such an exact formula is an impossible task, since time needed for an addition, subtraction, and so on, depends an numbers being added, subtracted, and so on.
- The value of Tp(n) for any given n can be obtained only experimentally.
- Even with the experimental approach, one could face difficulties.
- In a multi user system the execution time of a program p depends on the number of other programs running on the computer at the time program p is running.

Contd..

- As there were some problems in determining the execution time using earlier methods, we will go one step further and count only the number of *program steps.*

program step

program step is *loosely* defined as a syntactically or semantically meaningful program *segment* whose execution time is independent of the *instance characteristics.*

- Example

```
result = a + b + b * c + (a + b - c) / (a + b) + 4.0;
sum = a + b + c;
```

- The number of steps assigned to any *program statement* depends on the kind of statement.
- Comments are counted as zero number of steps.
- An assignment statement which does not involve any calls to other functions counted as one step.
- Loops, such as the for, while, and repeat-until, we consider the step counts only for the control part of the statement.
- The control parts for for and while statements have the following forms:

for i:= <expr1> to <expr2> do

while (<expr>) do

- Each execution of the control part of a while statement is one, unless <expr> is a function of instance characteristics.
- The step count for each execution of the control part of a for statement is one, unless <expr1> and <expr2> are functions of the instance characteristics.
- The step count for each execution of the condition of a conditional statements is one, unless condition is a function of instance characteristics.
- If any statement (assignment statement, control part, condition etc.) involves function calls, then the step count is equal to the number of steps assignable to the function plus one.

- Methods to compute the step count

  1) Introduce global variable count into programs with initial value zero.
     - Statements to increment count by the appropriate amount are introduced into the program.
     - The change in the value of the count by the time this program terminates is the number steps executed by the program.

  2) Tabular method
     - Determine the total number of steps contributed by each statement per execution × frequency
     - Add up the contribution of all statements

Method-I: Introduce variable count into programs

EX:- Iterative sum of n numbers

```
Algorithm sum(a, n)
{
        s:=0;
          count:=count+1;  // for assignment statement
          for i:=1 to n do
          {     count:=count+1; // For for
                s:=s+a[i];
                count:=count+1; // for assignment statement
          }
              count:=count+1; // for last time of for
           return s;
        count:=count+1; // for return
  }
```

2n + 3 steps

Note : *Step count tells us how the run time for a program changes with changes in the instance characteristics.*

Ex:- Addition of two m×n matrices

```
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];

                }
    }

 }
```

2mn + 2m+1 steps

EX:- Recursive sum of n numbers

```
Algorithm RSum(a,n)
{
    count:=count+1; // for the if conditional
    if(n ≤ 0) then
    {
            return 0;
           count:=count+1; // for the return
    }
    else
    {
            return RSum(a,n-1)+a[n];
          count:=count+1;   // For the addition, function invocation and return
    }
}
```

- When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count.
- We obtain the following recursive formula for above (RSum) algorithm.

$$t_{RSum}(n) = \begin{cases} 2 & \text{If } n=0 \\ 2 + t_{RSum}(n-1) & \text{If } n>0 \end{cases}$$

- One way of solving such recursive formula is by repeated substitutions for each occurrence of the function t_{RSum} on the right side until all such occurrences disappear:

$t_{RSum}(n) = 2+t_{RSum}(n-1)$
$= 2+2+t_{RSum}(n-2)$
$= 2(2)+t_{RSum}(n-2)$
$= 2+2+2+_{tRSum}(n-3)$
$= \mathbf{2(3)+t_{RSum}(n-3)}$
:
:
$= n(2) +t_{RSum}(n-n)$
$=2n+t_{RSum}(0)$
$= 2n+2$

The step count for Rsum is 2n+2

Definition : Input size

- One of the instance characteristics that is frequently used in the literature is the input size.

- The input size of any instance of a problem is defined to be number of words needed to describe that instance.

- The input size for the problem of summing and array of n elements is n+1,

  where n for listing the n elements and

  1 for the value of n.

- EX:- Iterative sum of n numbers

Method-II: Tabular method

Statement	s/e	frequency	Total steps

Algorithm sum(a, n)
{
s:=0 ;
for i:=1 to n do
s:=s+a[i];
return s;
}
	0
	0
	1
	1
	1
	1
	0
		--
		--
		1
		n+1
		n
		1
		--
			0
			0
			1
			n+1
			n
			1
			0

Total			2n+3

- EX:- Addition of two m×n matrices

Statement	s/e	frequency	Total steps

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] ;
}
	0
	0
	1
	1
	1
	0
		--
		--
		m+1
		m(n+1)
		mn
		--
			0
			0
			m+1
			mn+m
			mn
			0

Total			2mn+2m+1

- EX:- Recursive sum of n numbers

Statement	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithm RSum(a,n)	0	--	--	0	0
{	0	--	--	0	0
if(n ≤ 0) then	1	1	1	1	1
return 0;	1	1	0	1	0
else return					
Rsum(a,n-1)+a[n] ;	1+x	0	1	0	1+x
}	0	--	--	0	0
Total				2	2+x

$x=t_{RSum}(n-1)$

Best, Worst, Average Cases

- ✓ Not all inputs of a given size take the same number of program steps.
- ✓ Sequential search for *K* in an array of *n* integers:
  - Begin at first element in array and look at each element in turn until *K* is found.

1. Best-Case Step count:-

   Minimum number of steps executed by the algorithm for the given parameters.

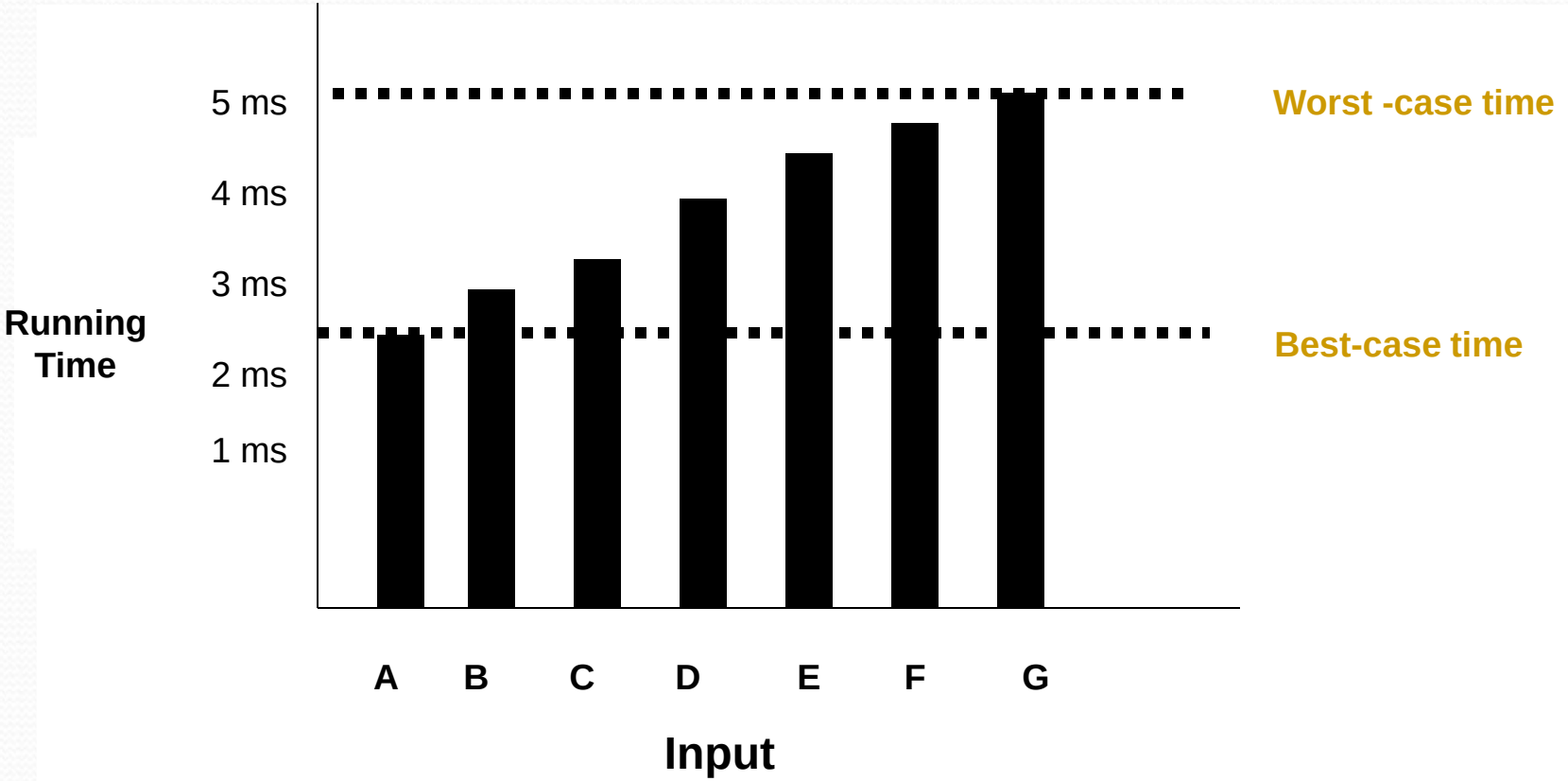
2. Worst-Case Step count:-

   Maximum number of steps executed by the algorithm for the given parameters.

3. Average-Case Step count:-

   Average number of steps executed by an algorithm.

Contd..



$$\text{Average-case time} = \frac{\text{A time + B time+ ……..+G time}}{\text{7 (total number of possible inputs)}}$$

Inexactness of step count.

- Both the instructions x=y; and x=y+z+(x/y) count as one step.
- Therefore, two analysts may arrive at $4n^2+6n+2$ and $7n^2+3n+4$ as the step count for the same program.
- Because of the inexactness of what a step count stands for, the exact step count is not very useful for comparison of algorithms.

Asymptotic efficiency

- Asymptotic efficiency means study of algorithms efficiency for large inputs.

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a rough measure that characterizes how fast each function grows as n grows.

- *Hint:* use *rate of growth*
- Compare functions **asymptotically!**
  (i.e., for large values of n)

Rate of Growth

- Ex:- $f(n)=n^2+100n+\log_{10}n+1000$

n	f(n)	n2		100n		log10n		1000	
	value	value	%	value	%	value	%	value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
1,00000	10,010,001005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

- The low order terms and constants in a function are relatively insignificant for **large** n

$$n^2 + 100n + \log_{10}n + 1000 \quad \sim \quad n^2$$

i.e., we say that $n^2 + 100n + \log_{10}n + 1000$ and n^2 have the same **rate of growth**

Some more examples

- $n^4 + 100n^2 + 10n + 50$ is $\sim$?
- $10n^3 + 2n^2$ is $\sim$?
- $n^3 - n^2$ is $\sim$?
- constants
  - 10 is $\sim$?
  - 1273 is $\sim$?

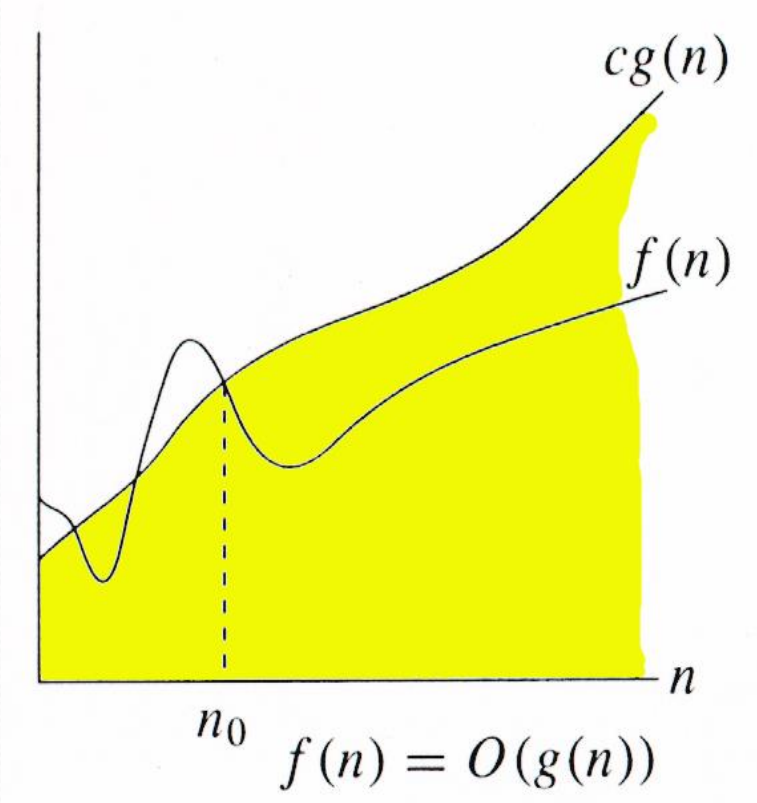
Asymptotic/order Notations

- Asymptotic/order notation describes the behavior of functions for the large inputs.
- **1. Big Oh(*O*) notation:**
  - The big oh notation describes an upper bound on the asymptotic growth rate of the function f.

Definition: [Big "oh"]

- f(*n*) = O(g(*n*)) (read as "*f* of *n* is big oh of *g* of *n*") iff there exist positive constants c and n_o such that

  $f(n) \leq cg(n)$ for all n, $n \geq n_o$.



- *In other words,* $g(n)$ grows faster than or same rate as f(n).
- **Examples**

  - $f(n) = 3n+2$
    - $3n + 2 <= 4n$, for all $n >= 2$, $\therefore 3n + 2 = O(n)$
  - $f(n) = 10n^2+4n+2$
    - $10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore 10n^2+4n+2 = O(n^2)$
  - $f(n)=6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

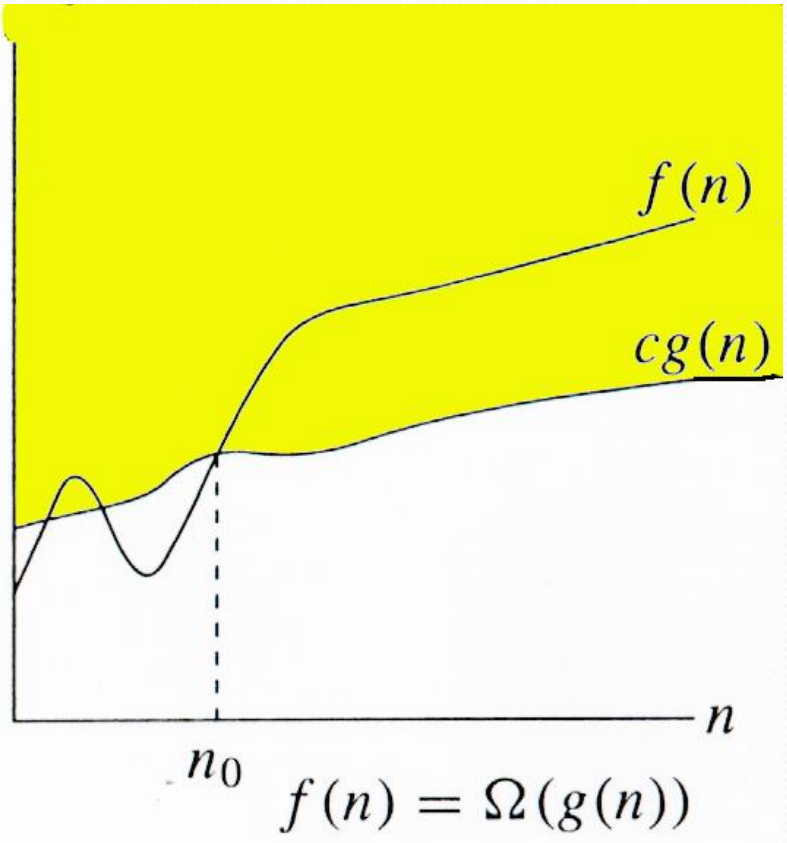
- It also possible to write $10n^2+4n+2 = O(n^3)$ since $10n^2+4n+2 <= 7n^3$ for $n>=2$

- Although n^3 is an upper bound for $10n^2+4n+2$, it is not a tight upper bound; we can find a smaller function (n^2) that satisfies big oh relation.

- But, we can not write $10n^2+4n+2 = O(n)$, since it does not satisfy the big oh relation for sufficiently large input.

2. *Omega* (Ω) notation:

- The omega notation describes a lower bound on the asymptotic growth rate of the function f.



Definition: [Omega]

- $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_o such that $f(n) \geq cg(n)$ for all n, $n \geq n_o$.

- *In other words, $f(n)$* grows faster than or same rate as g(n).

- **Examples**

  - $f(n) = 3n+2$

    - $3n + 2 >= 3n$, for all $n >= 1$, $\therefore 3n + 2 = \Omega(n)$

  - $f(n) = 10n^2+4n+2$

    - $10n^2+4n+2 >= n^2$, for all $n >= 1$, $\therefore 10n^2+4n+2 = \Omega(n^2)$

- It also possible to write $10n^2+4n+2 = \Omega(n)$ since $10n^2+4n+2 >= n$ for $n>=0$

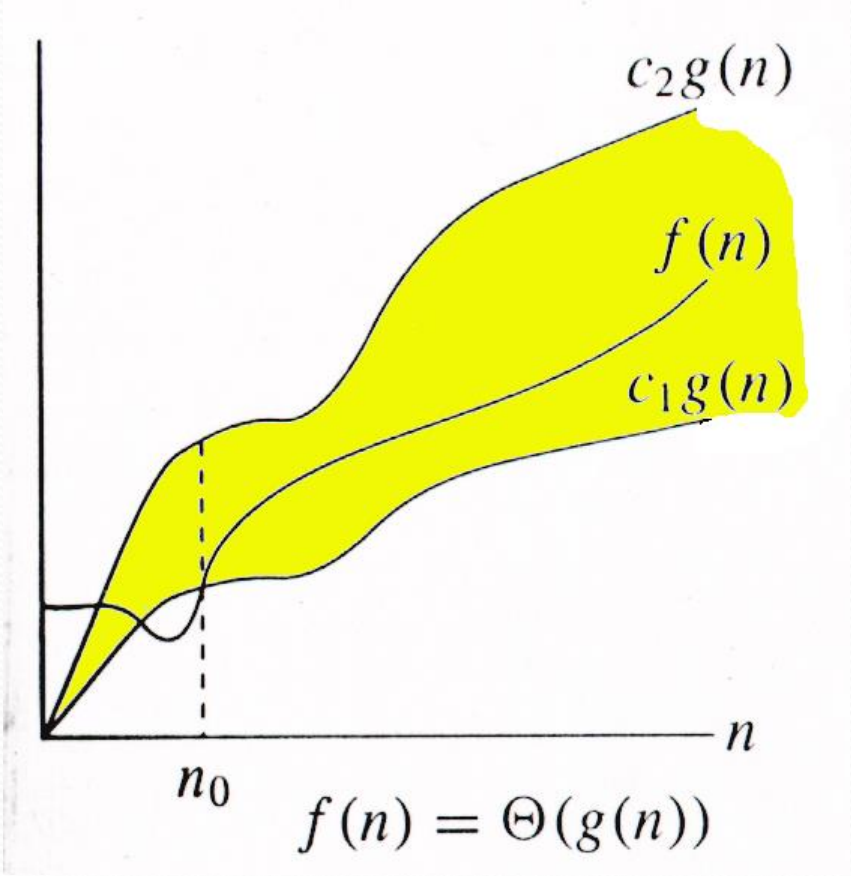
- Although n is a lower bound for $10n^2+4n+2$, it is not a tight lower bound; we can find a larger function (n^2) that satisfies omega relation.

- But, we can not write $10n^2+4n+2 = \Omega(n^3)$, since it does not satisfy the omega relation for sufficiently large input.

- ***3.Theta*** **(Θ) notation:**
  - The Theta notation describes a tight bound on the asymptotic growth rate of the function f.

Definition: [Theta]

- $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.



- *In other words, f(n) grows same rate as g(n).*
- *Examples:-*
  - $f(n) = 3n+2$
    - $3n <= 3n + 2 <= 4n$, for all $n >= 2$, $\therefore 3n + 2 = \Theta(n)$
  - $f(n) = 10n^2+4n+2$
    - $n^2 <= 10n^2+4n+2 <= 11n^2$, for all $n >= 5$,

      $\therefore 10n^2+4n+2 = \Theta(n^2)$

Note: we can not write either $10n^2+4n+2 = \Theta(n)$ or $10n^2+4n+2 = \Theta(n^3)$, since neither of these will satisfy the theta relation.

4. Little Oh(o) notation:

- The big oh notation describes a strict upper bound on the asymptotic growth rate of the function f.

Definition: [Little "oh"]

- f(n) = o(g(n)) (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- The definition states that the function f(n) is less than c times the function g(n) except when n is smaller than n_o.
- *In other words, f(n) grows slower than" g(n).*
- **Examples**
  - $f(n) = 3n+2=o(n^2)$
    - Since

$$\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$$

  - *However, $3n+2 \neq o(n)$*

Big-Oh, Theta, Omega and Little-oh

Tips :

- Think of O(g(n)) as "less than or equal to" g(n)
  - Upper bound: "grows slower than or same rate as" g(n)
- Think of Ω(g(n)) as "greater than or equal to" g(n)
  - Lower bound: "grows faster than or same rate as" g(n)
- Think of Θ(g(n)) as "equal to" g(n)
  - "Tight" bound: "same growth rate"
- Think of o(g(n)) as "less than to" g(n)
  - Strict Upper bound: "grows slower than " g(n)
-

Functions ordered by growth rate

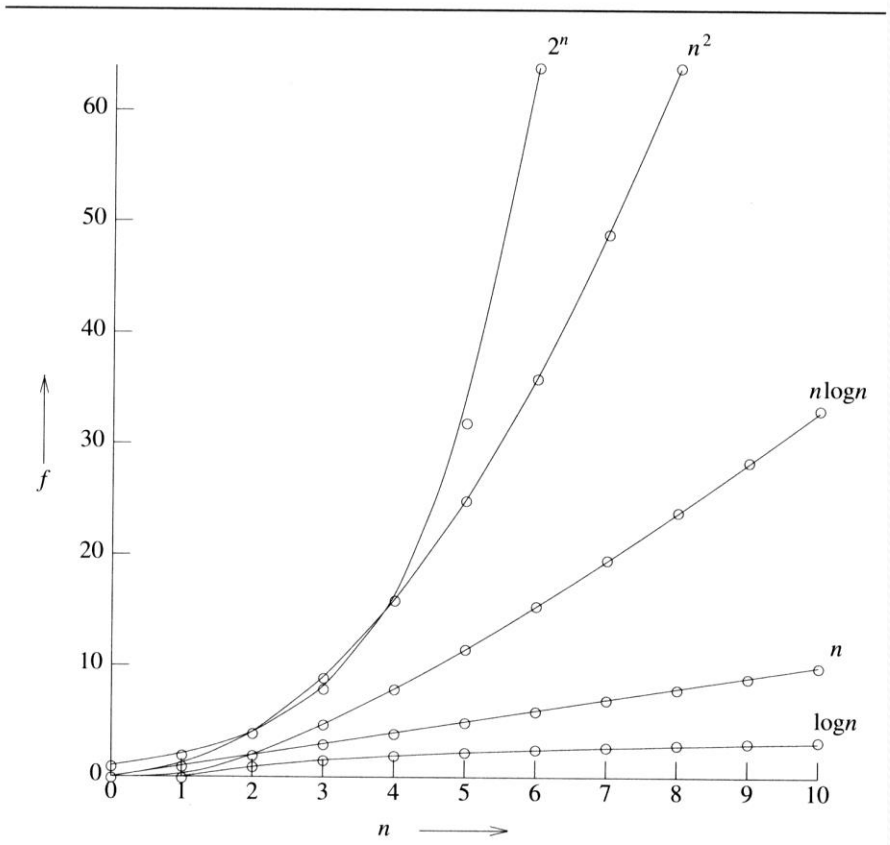
Function	Name
1	Growth is constant
logn	Growth is logarithmic
n	Growth is linear
nlogn	Growth is n-log-n
n^2	Growth is quadratic
n^3	Growth is cubic
2^n	Growth is exponential
n!	Growth is factorial

$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$

- To get a feel for how the various functions grow with n, you are advised to study the following figs:

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values



2

- The following fig gives the time needed by a 1 billion instructions per second computer to execute a program of complexity $f(n)$ instructions.

Times on a 1 billion instruction per second computer

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years


THANK
YOU!


KEEP
CALM
AND
SAVE
TREES
KeepCalmAndPosters.com