MVJ22CS41 – ANALYSIS AND DESIGN OF ALGORITHMS

Module 1

Semester – IV Section - A

Academic Year : 2024-2025(EVEN)

By

NIKITHA G S

PREREQUISITES:

BASIC KNOWLEDGE ABOUT ANALYSIS AND DESIGN OF ALGORITHMS

BRIDGE MATERIAL:

Algorithm is a sequence of steps to solve a problem. Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems. As the speed of processor increases, performance is frequently said to be less central than other software quality characteristics (e.g. security, extensibility, reusability etc.).

ANALYSIS AND DESIGN OF ALGORITHM SUBJECT CODE : MVJ22CS41

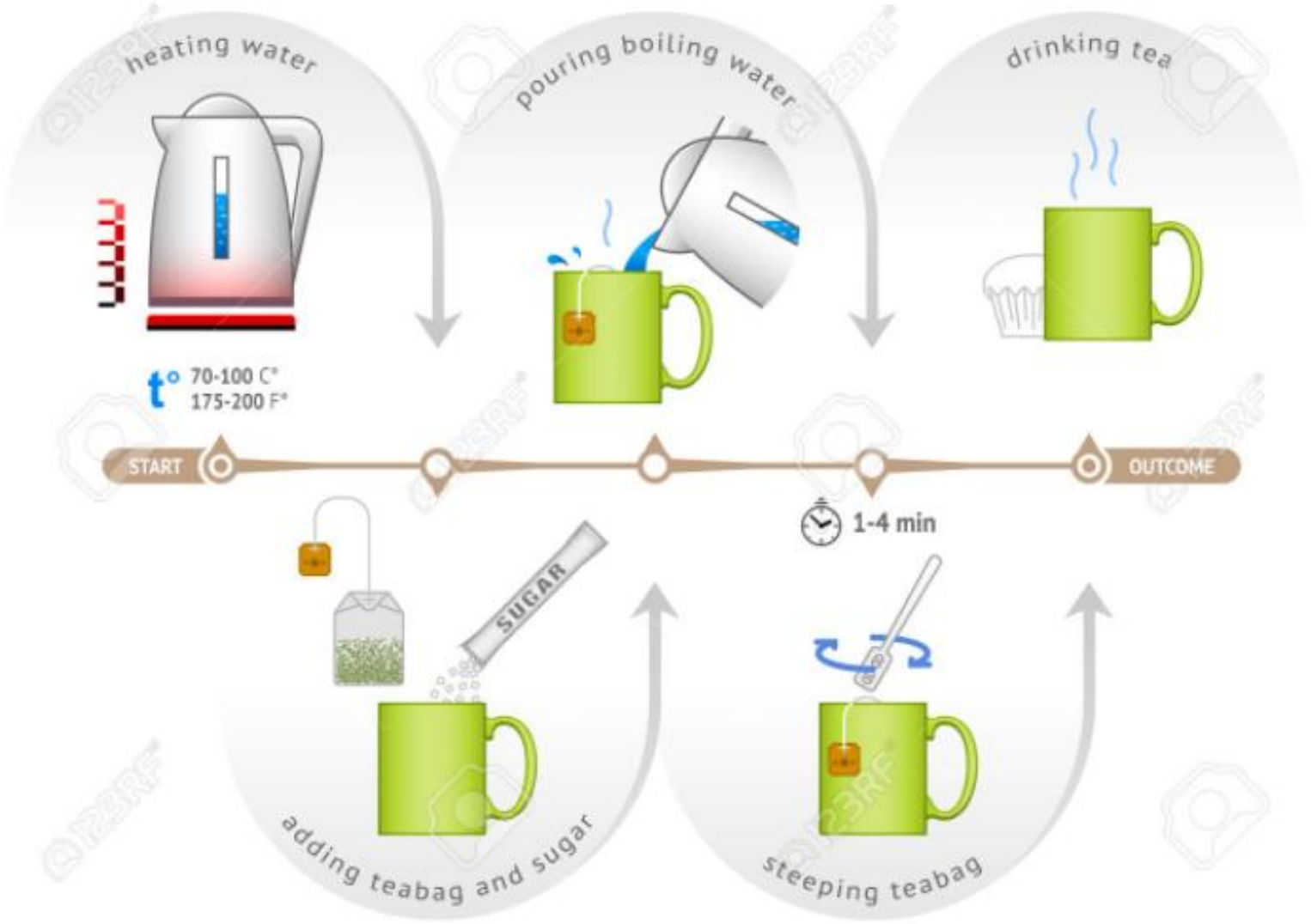
Basic Concept of Algorithms: Introduction-What is an Algorithm, Algorithm Specification, Analysis Framework, Performance Analysis: Space complexity, Time complexity. Asymptotic Notations: Big-Oh notation (O), Omega notation (Ω), Theta notation (Θ), and Little-oh notation (o), Mathematical analysis of Non-Recursive and recursive Algorithms with Examples. Important Problem Types. Fundamental Data Structures

Applications: developing computational tools and bioinformatics software, Mathematics

Video link / Additional online information (related to module if any):

1. http://www.nptelvideos.com/video.php?id=1442
2. https://nptel.ac.in/courses/106105085/

Tea Preparation

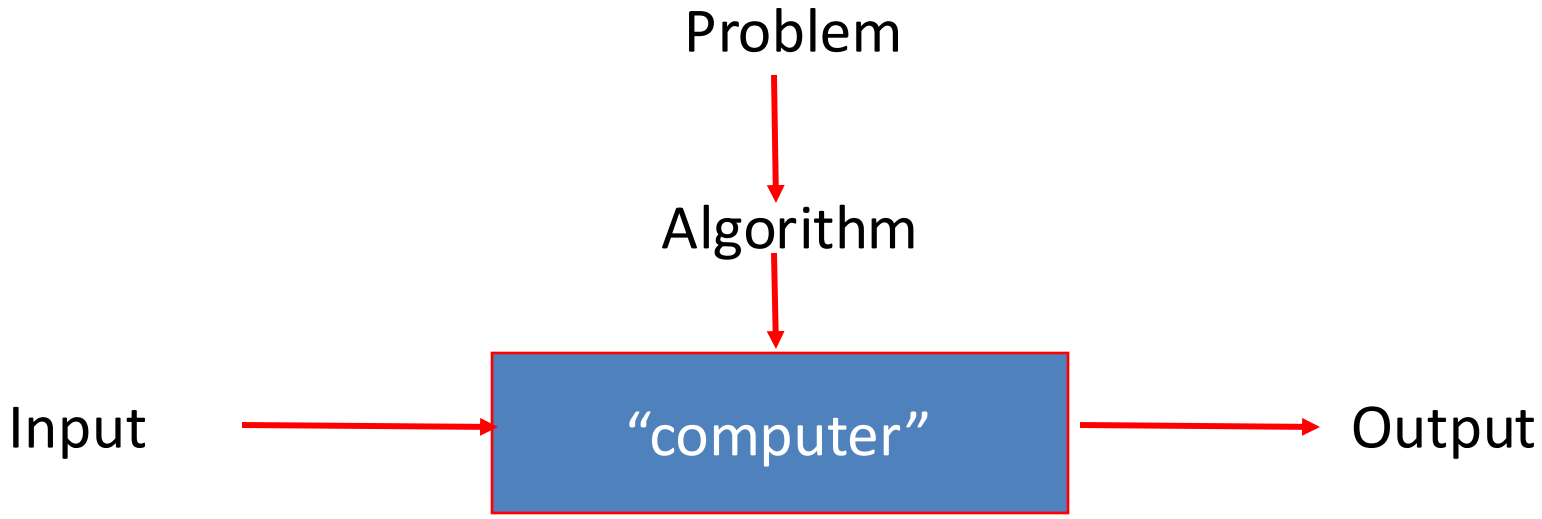


Noodles Preparation

WHAT IS AN ALGORITHM?

An algorithm is a step by step procedure for solving a computational problem .

An 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.



Advantages of Algorithms:

It is easy to understand.

Algorithm is a step-wise representation of a solution to a given problem.

In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Example :

Suppose, we have to make an analysis of the stock market and we have data for one month. Our analysis needs us to find the day on which a maximum profit could be made. Let's assume that the investment will be done at the start of the day and it will be sold at the end. Of course, this isn't entirely a practical problem, at least with the assumptions we have made but it will make you understand what basically an algorithm is.

Day	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Price at Start of Day	45	50	48	52	35	39	36	40	42	35
Price at End of Day	50	48	52	35	39	36	40	42	35	41

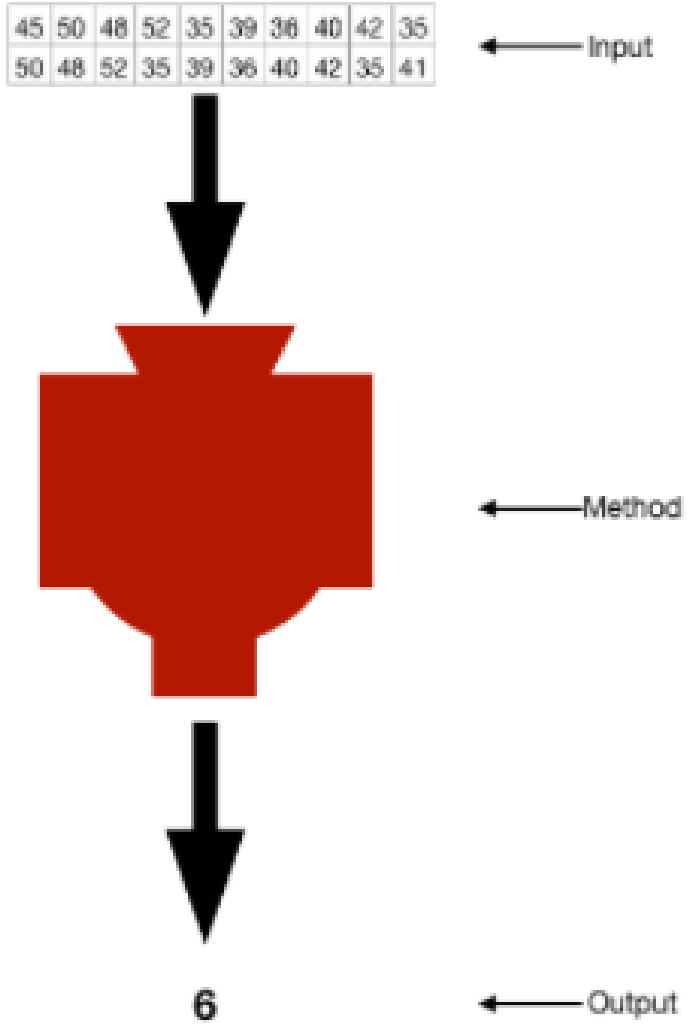
So, we have a problem with the data for one month and this is the **input** to our problem. Our problem also has some constraints about when a stock can be purchased and when it will be sold. And the expected **output** is the maximum profit.

To solve this problem, we would first calculate the difference between the prices of the stocks at the end of the day and at the start of the day. Then we will calculate the maximum among these differences and that would be our output.

Day	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Price at Start of Day	45	50	48	52	35	39	36	40	42	35
Price at End of Day	50	48	52	35	39	36	40	42	35	41
Difference	5	-2	4	-17	4	-3	4	2	-7	6

Maximum = 6 at Day 10

So, we have basically developed a **method** to solve a **specific problem** and our method takes an **input (data of one month)** and gives us an **output (the day of maximum profit)** and this is an ***algorithm***.



Characteristics of Algorithm:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

All algorithms must satisfy the following criteria:

Input. An algorithm has zero or more inputs, taken from a specified set of objects.

Output. An algorithm has one or more outputs, which have a specified relation to the inputs.

Definiteness. Each step must be precisely defined; Each instruction is clear and unambiguous.

Finiteness. The algorithm must always terminate after a finite number of steps.

Effectiveness. All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

ALGORITHM SPECIFICATION

We can describe an algorithm in many ways.

We can use a natural language like English
Graphic representations called flowcharts are another possibility

1. Comments begin with // and continue until the end of line.
2. Blocksare indicatedwith matchingbraces:{ and }
3. An identifier begins with a letter

```
node= record
{
datatypeA dataA;
datatype_n data_n;
node *link;
}
```

link is a pointer to the record type node. Individual data itemsof a record can be accessed with -> and period. For instance if p points to a record of type node,p-> dataA
if q is a record of type node, q.data will denote its first field

4. Assignment of values to variables is done using the assignment statement (variable):=(expression);

5.There are two Boolean values true and false. In order to produce these values, the logical operators and,or,and not and the relational operators<, ≤, = , ≠, ≥ and > are provided.

6.Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i,j) the element of the array is denoted as A[i,j].Array indices start at zero.

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

The while loop takes the following form:

```
while (condition) do
 {
(statement 1)
……
…..
(statement n)
}
```

The general form of a for loop is

for variable := valuel to value2 **step** step do
{
(statement 1)
……..
……..
(statement n)
}

```
variable:=valuel;
fin :=value2;
incr :=step;
while ((variable - fin) * step<0) do
{
(statement1)
……
(statementn)
variable:=variable+ incr;
}
```

A repeat-until statement is constructed as follows:

```
repeat
(statement1)
…….
…….
(statementn)
until(condition)
```

The instruction break; can be used within any of the above looping instructions to force exit.

8. A conditional statementhas the following forms:
if (condition) then (statement)
if (condition) then (statement1) else (statement2)

Here(condition) is a boolean expression and (statement),(statement1) and (statement2) are arbitrary statements (simple or compound).

We also employ the following case statement:
case
{
:(condition 1):(statement1)
……..
……..
:(conditino n) : (statementn)
:else:(statementn + 1)
}

9. Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities

10.There is only one type of procedure: Algorithm An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name((parameterlist))

Where Name is the name of the procedure and ({parameterlist)) is a listing of the procedure parameters

The body has one or more (simple or compound)statements enclosed within braces {and }.

Write an algorithm to find the maximum number of an array A

```
1 Algorithm Max(A, n)
2 // A is an array of size n.
3 {
4 Result:=A[0];
5 for i :=1 to n do
6 if A[i] >Result then Result:=A[i];
7 return Result;
8 }
```

Write an algorithm to count the sum of n numbers.

Problem Description: Algorithm to find the sum of n numbers.
Input: 1 to n numbers
Output: Sum of n numbers

result:<- 0
for i<-1 to n do i<-i+1
Result<-result+i;
return result

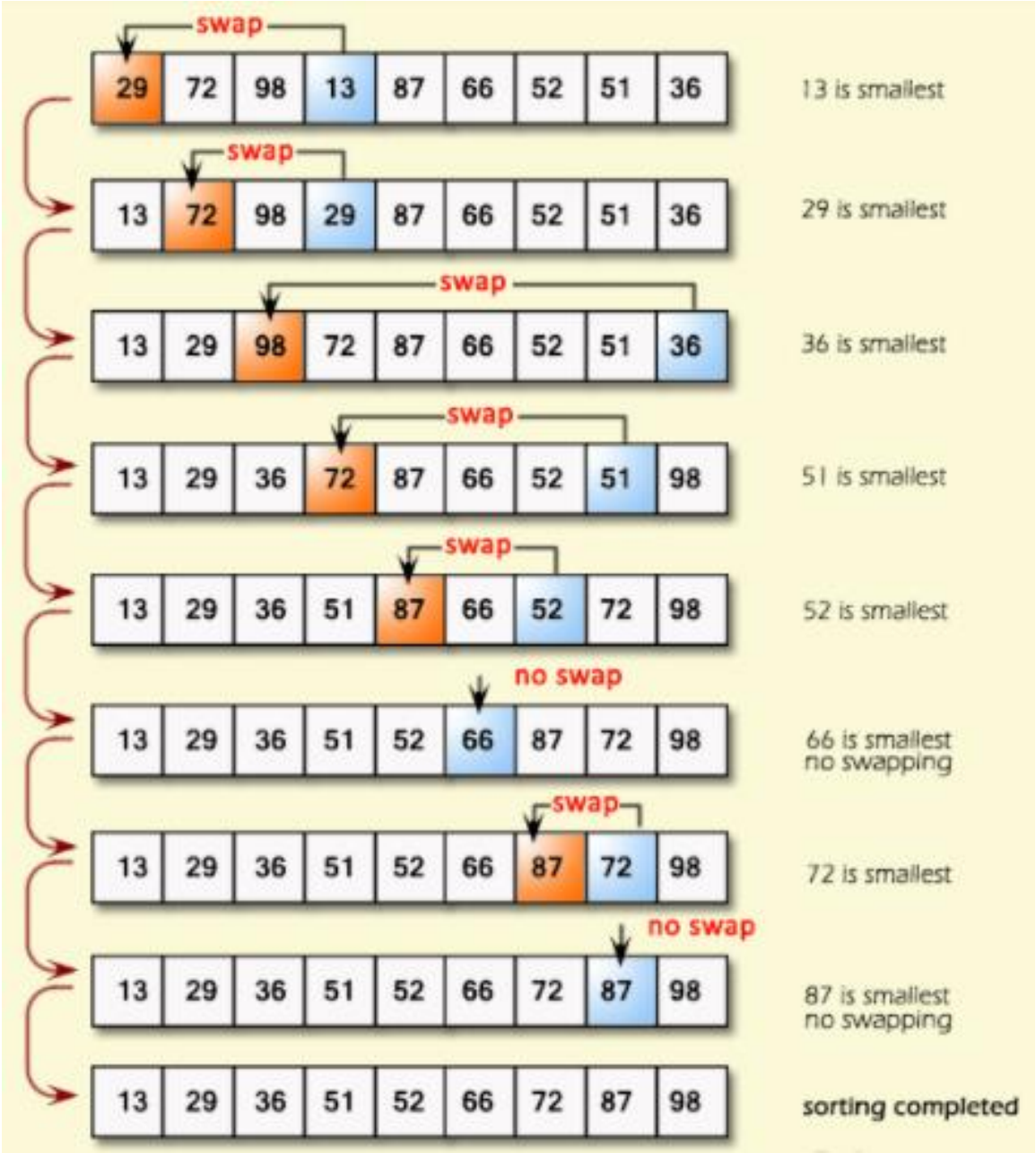
Write an algorithm to check whether the given number is even or odd.

Problem Description: Algorithm to **test whether the given number is even or odd.**
Input: The number val, to be tested
Output: Message indicating even or odd

if(val%2=0) then
write("Given number is even")
else
write("Given number is odd")

Selection sort algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning



Example: Algorithm SelectionSort(a,n) correctly sorts a set of n >= 1 elements; the result remains in a[1:n] such that a[l]<= a[2]<= a[3]…….<=a[n].

Example1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of n >1 elements of arbitrary type.

```
for (i=1; i<=n; i++) {
   examine a[i] to a[n] and suppose
   the smallest element is at a[j];
   interchange a[i] and a[j];
}
```

```
Algorithm SelectionSort(a,n)
// Sort the array a[1:n] into non decreasing order.
{
for i <-1to n-1 do
{
min <-i
for j<-i+ 1 to n do
if (a[min]>a[j]) then min<-j;
if(min≠i)
{
t <-a[i];
a[i]<-a[min]
a[min]<-t;
} } }
```

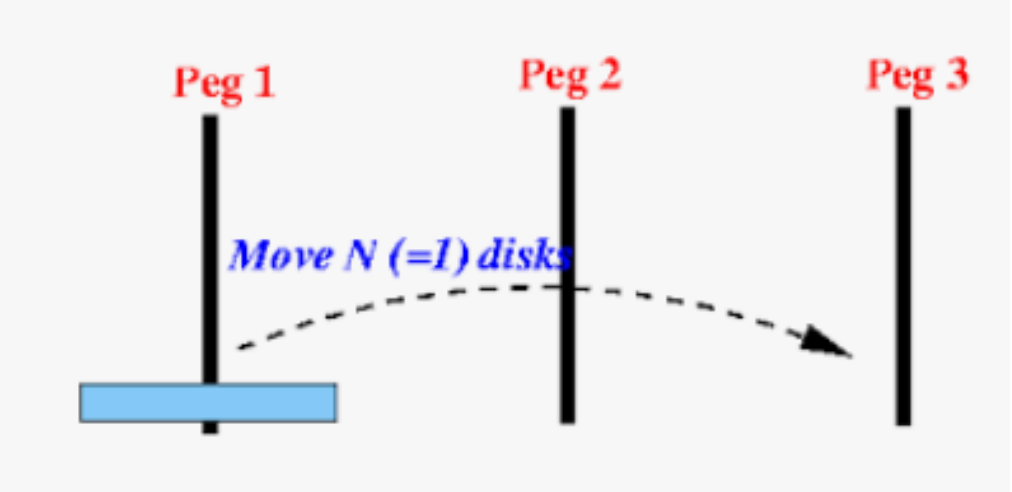
Recursive algorithms

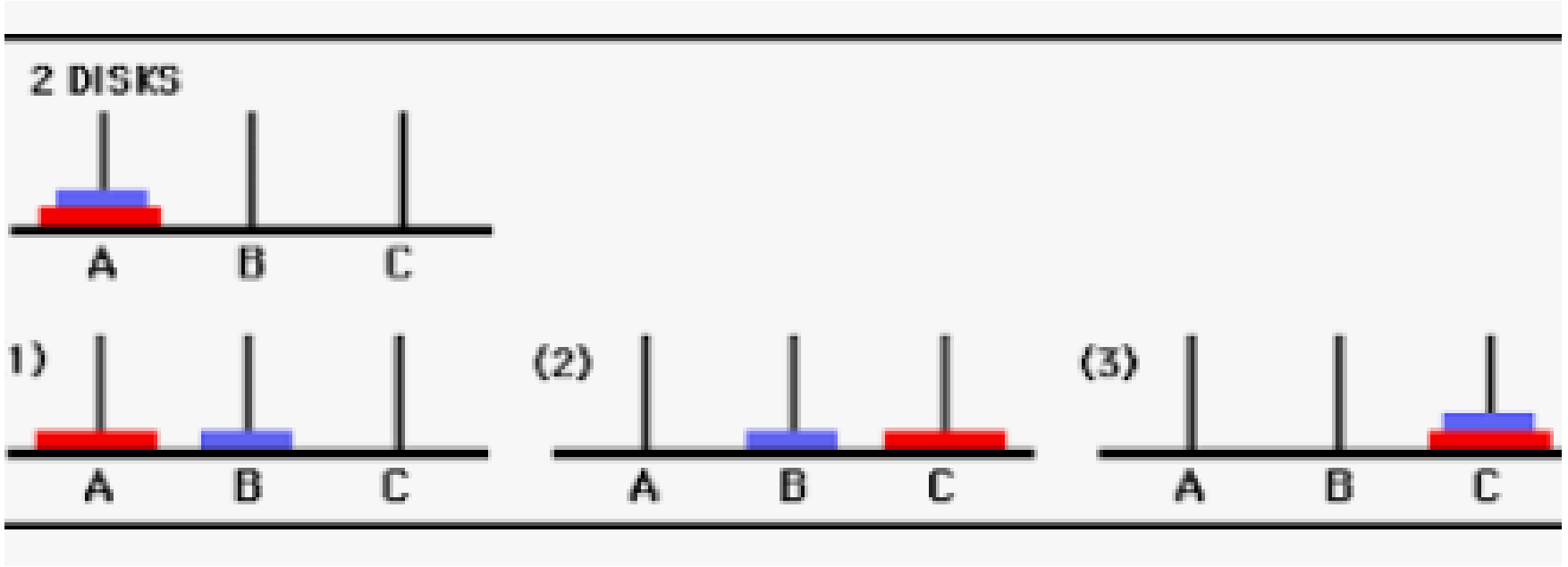
An algorithm is said to be **recursive** if the same algorithm is invoked in the body (direct recursive).

Algorithm ***A*** is said to be **indirect recursive** if it calls another algorithm which in turn calls A.

Example 1: Tower of Hanoi

- How to Solve: Strategy…
  - Generalize first: Consider n disks for all n ≥ 1
  - Our example is the case when n=3
- Look at small instances…
  - How about n=1
    - Of course, just "Move disk 1 from A to C"
  - How about n=2?
    1. "Move disk 1 from A to B"
    2. "Move disk 2 from A to C"
    3. "Move disk 1 from B to C"





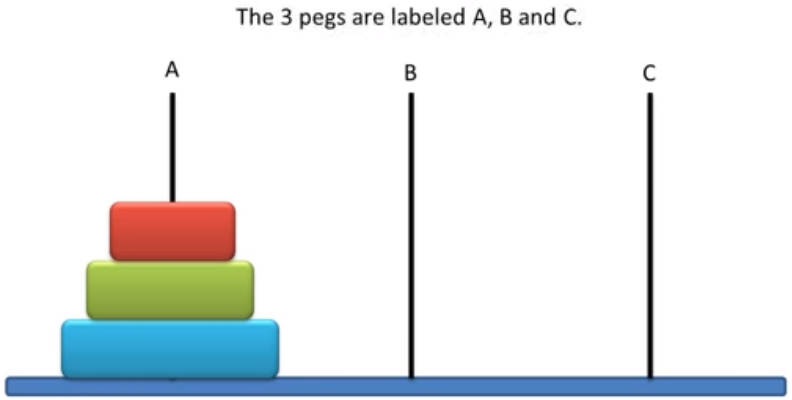
1. “Move disk 1 from A to B”
2. “Move disk 2 from A to C”
3. “Move disk 1 from B to C”

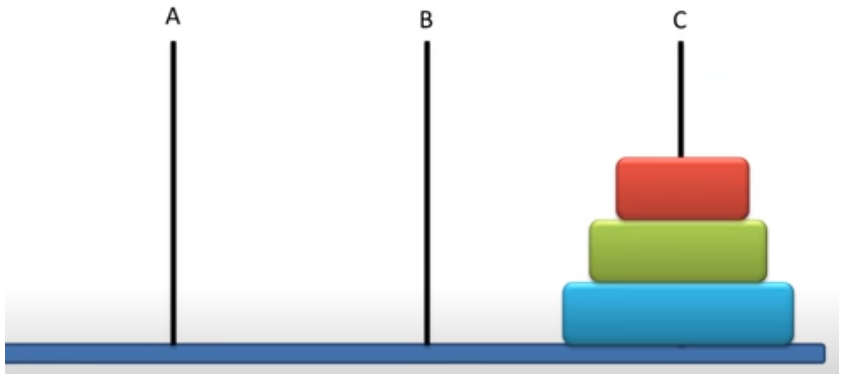
- General Method:
  - First, move first (n-1) disks from A to B
  - Now, can move largest disk from A to C
  - Then, move first (n-1) disks from B to C
- Try this method for n=3

1. “Move disk 1 from A to C”
2. “Move disk 2 from A to B”
3. “Move disk 1 from C to B”

4. “Move disk 3 from A to C”

5. “Move disk 1 from B to A”
6. “Move disk 1 from B to C”
7. “Move disk 1 from A to C”





Given: Three Pegs A, B and C

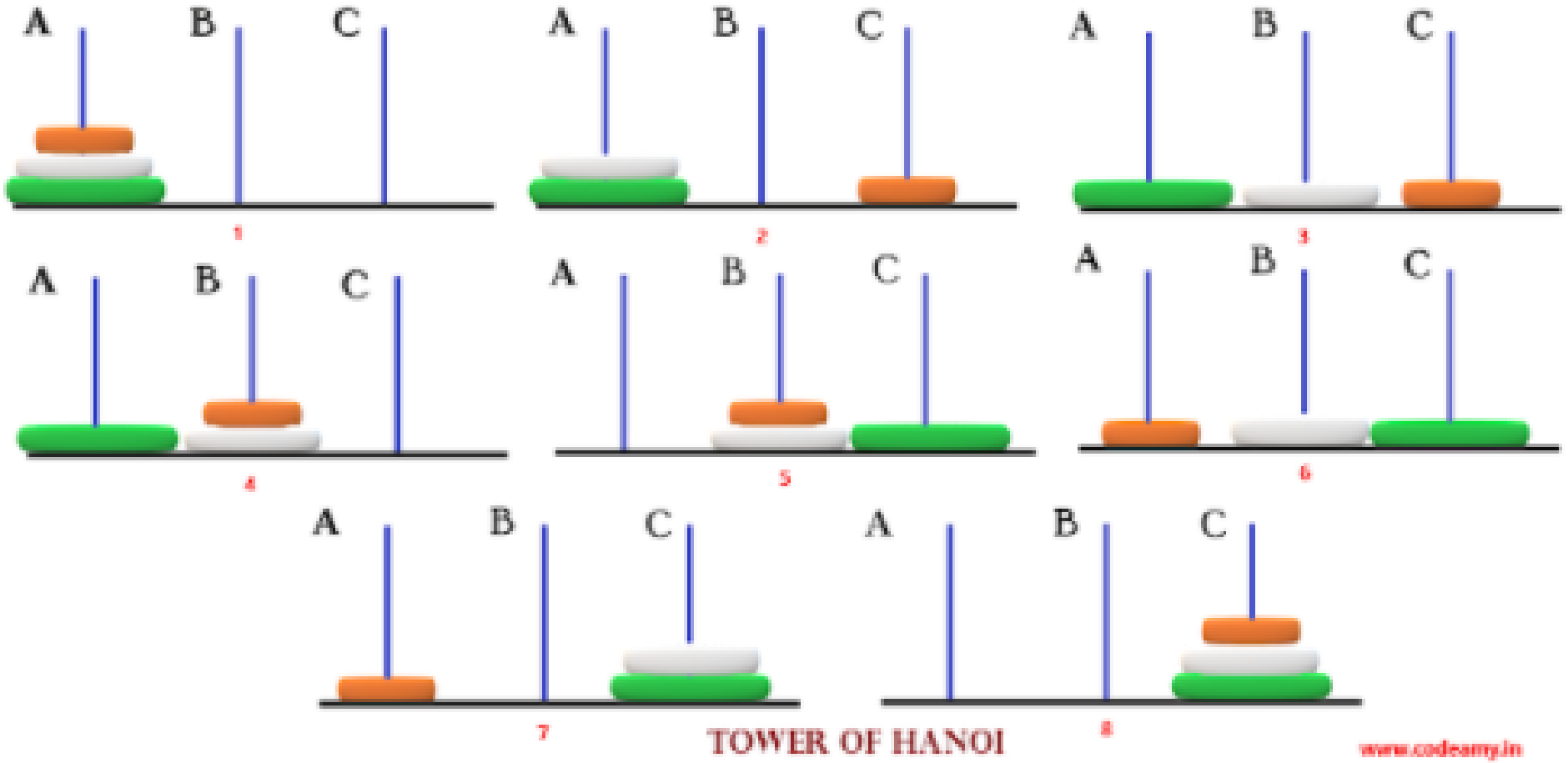
Peg A initially has n disks, different size, stacked up,

larger disks are below smaller disks

Problem: to move the n disks to Peg C, subject to the following

1. Only one disk moves at a time.
2. Only the "top" disk can be removed.
3. In each movement taking the top disk from one tower to top of another tower but you can not place larger on a smaller disk.
4. All disk should be moved from source to destination with few moves.

The below image shows the step by step movement of disks from one tower to another and it takes 7 moves for 3 disks



Procedure:

Step 1:

If n=1, Move disc from source to destination

Step 2:

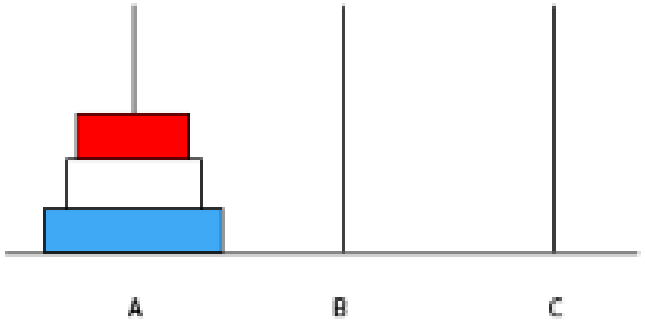
If n>1, Move n-1 disc to source to auxiliary using destination

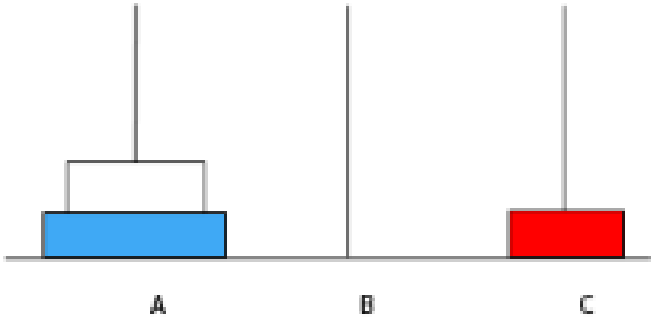
Step 3:

Move disc from source to destination

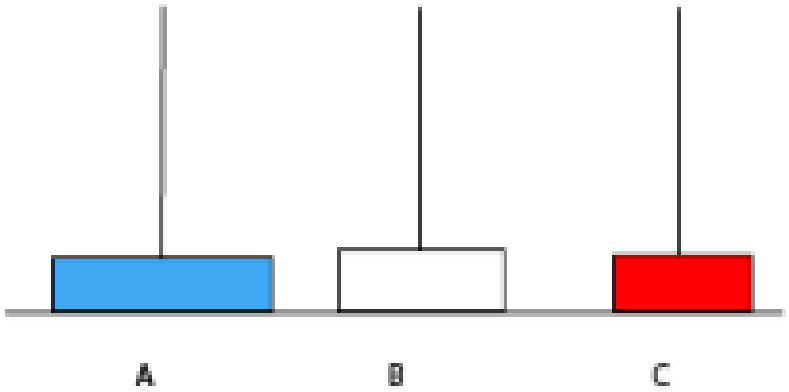
Step 4:

Move n-1 disc from auxiliary to destination using source

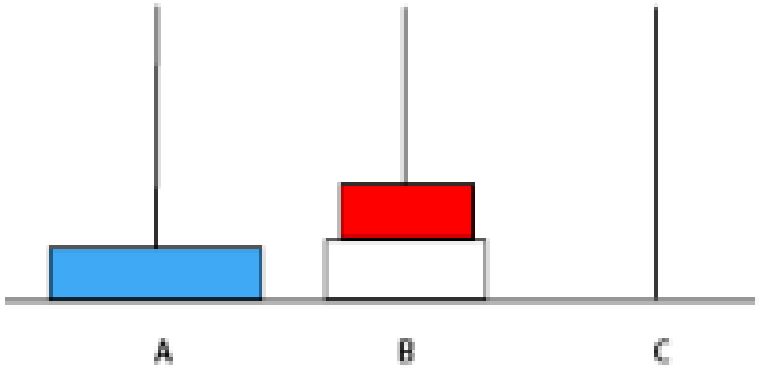




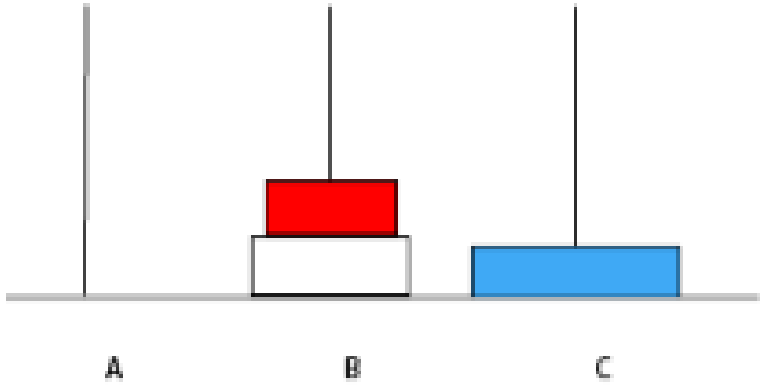
Disk 1 from A to C



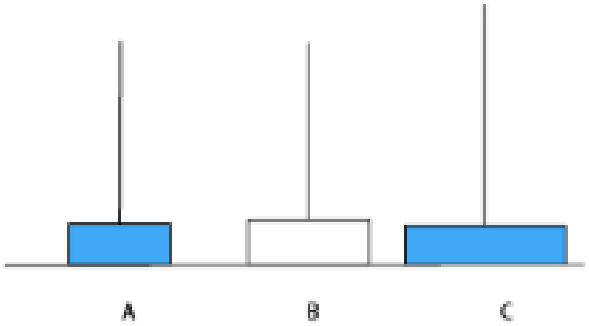
Disk 2 from A to B



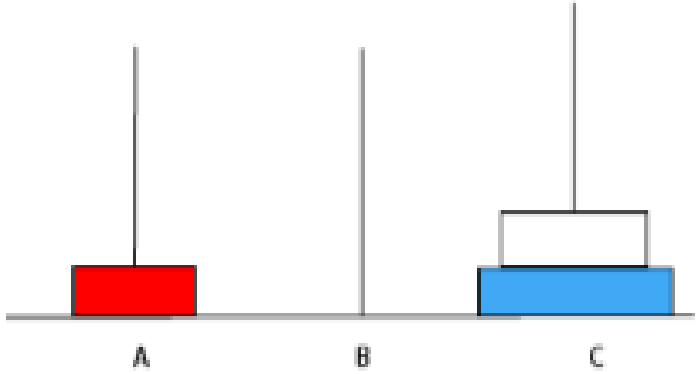
Disk 1 from C to B



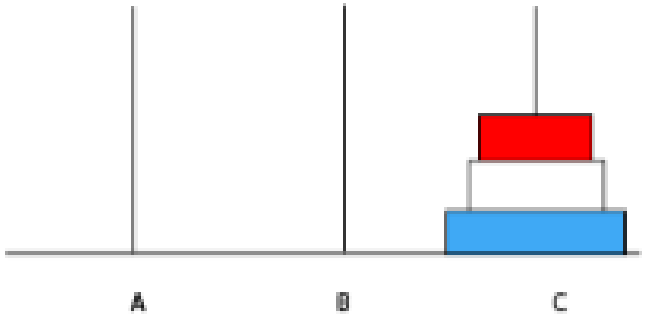
Disk 3 from A to C



Disk 1 from B to A



Disk 2 from B to C



Disk 1 from A to C

Algorithm for Towel of Hanoi (recursive)

- Recursive Algorithm
  - when (n=1), we have simple case
  - Else (decompose problem and make recursive-calls)

```
Hanoi(n, A, B, C);
(* Move n disks from A to C via B *)
begin
  if (n=1) then "Move top disk from A to C"
  else (* when n>1 *)
    Hanoi (n-1, A, C, B);
    "Move top disk from A to C"
    Hanoi (n-1, B, A, C);
  end if
end;
```

Step 1:
If n=1, Move disc from source to destination

Step 2:
If n>1, Move n-1 disc to source to auxiliary using destination

Step 3:
Move disc from source to destination

Step 4:
Move n-1 disc from auxiliary to destination using source

```
import java.io.*;
public class Main
{
public static void main(String args[])
{
 int nDisks = 3;
doTowers(nDisks, 'A', 'B', 'C');
 }
public static void doTowers(int nDisks, char source, char auxi, char desti)
{
if (nDisks == 1)
{
 System.out.println("Disk 1 from " + source + " to " + desti);
  }
 else {
doTowers(nDisks - 1, source, desti, auxi);//recursive function
System.out.println("Disk " + nDisks+ " from " + source + " to " + desti);
doTowers(nDisks - 1, auxi, source, desti);
 }                    }                    }
```

Tower of Hanoi

doTowers(nDisks, 'A', 'B', 'C');

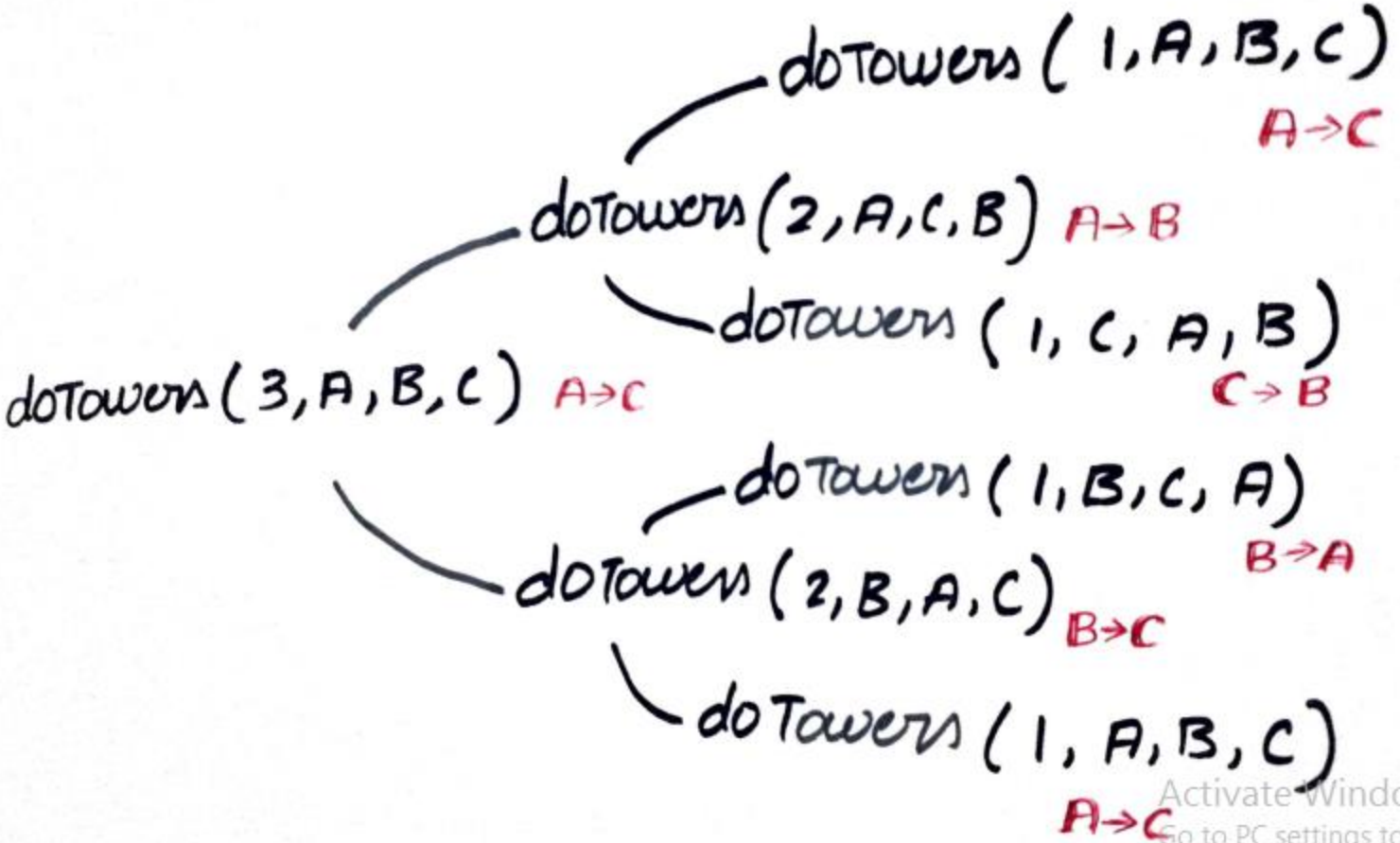
1.If nDisks=1, then write A->C, exit

2.(Move n-1 disks to source to Auxi)
doTowers(n-1, source,desti,auxi)

3.Write source->desti

4.(Move n-1 disks to Auxi to Desti)
doTowers(n-1, Auxi, source, desti)

5.Exit



Recursive permutation Generator

A **permutation** is a mathematical technique that determines the number of possible arrangements in a set when the order of the arrangements matters.

Common mathematical problems involve choosing only several items from a set of items with a certain order.

A **Permutation of a string** is another **string** that contains same characters, only the order of characters can be different. For example, “abcd” and “dabc” are **Permutation** of each other

When we are finding permutations for ‘TWO’, we are basically just deciding which letters to fill in three spaces (_ _ _). Now that that’s out of the way, this is how I would find all the permutations:

1.Start with the letter *T* and find the permutations that must start with *T*. Clearly the only permutations that start with *T* are ‘**TWO**’ and ‘**TOW**’. Add them to our list of permutations

2.Next take the letter *W* and find the permutations that start with it. This yields ‘**WTO**’ and ‘**WOT**’.

3.Finally look at *O*. The permutations that start with *O* are ‘**OTW**’ and ‘**OWT**’.

Recursive permutation Generator Algorithm :

Algorithm

1. Define a string.
2. Fix a character and swap the rest of the characters.
3. Call the generatePermutation() for rest of the characters.
4. Backtrack and swap the characters again.

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time

Program to find all the permutations of a string.

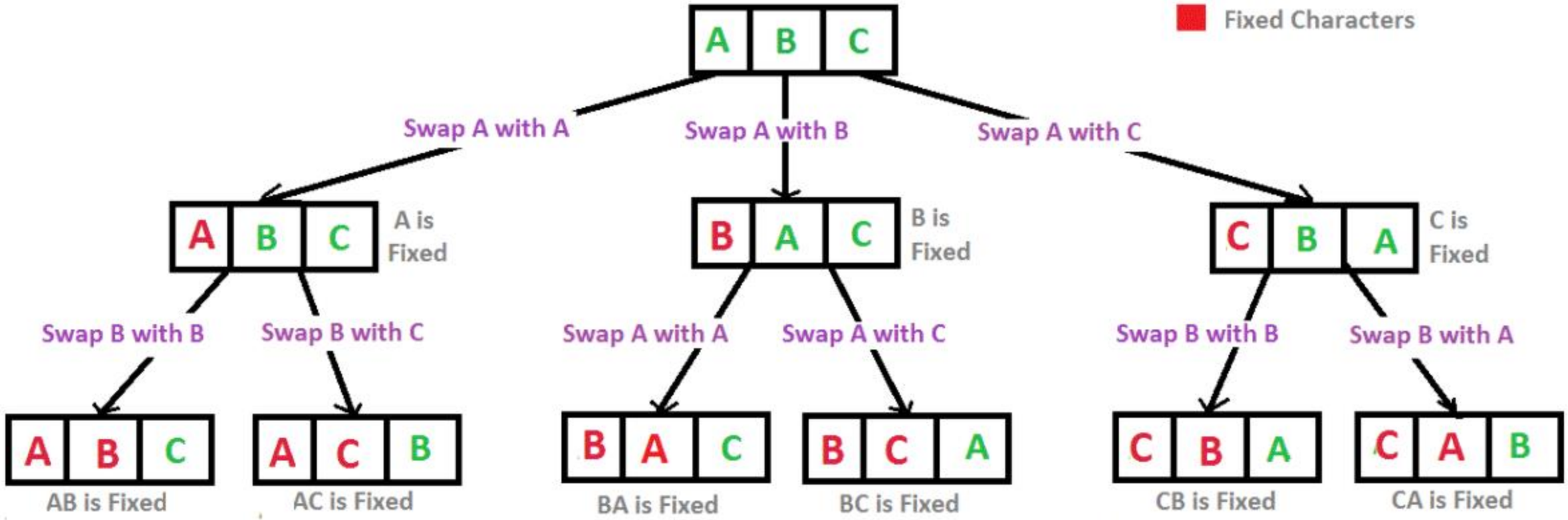
To solve this problem, we need to understand the concept of backtracking. According to the backtracking algorithm:

•Fix a character in the first position and swap the rest of the character with the first character. Like in ABC, in the first iteration three strings are formed: ABC, BAC, and CBA by swapping A with A, B and C respectively.

•Repeat step 1 for the rest of the characters like fixing second character B and so on.

•Now swap again to go back to the previous position. E.g., from ABC, we formed ABC by fixing B again, and we backtrack to the previous position and swap B with C. So, now we got ABC and ACB.

•Repeat these steps for BAC and CBA, to get all the permutations.

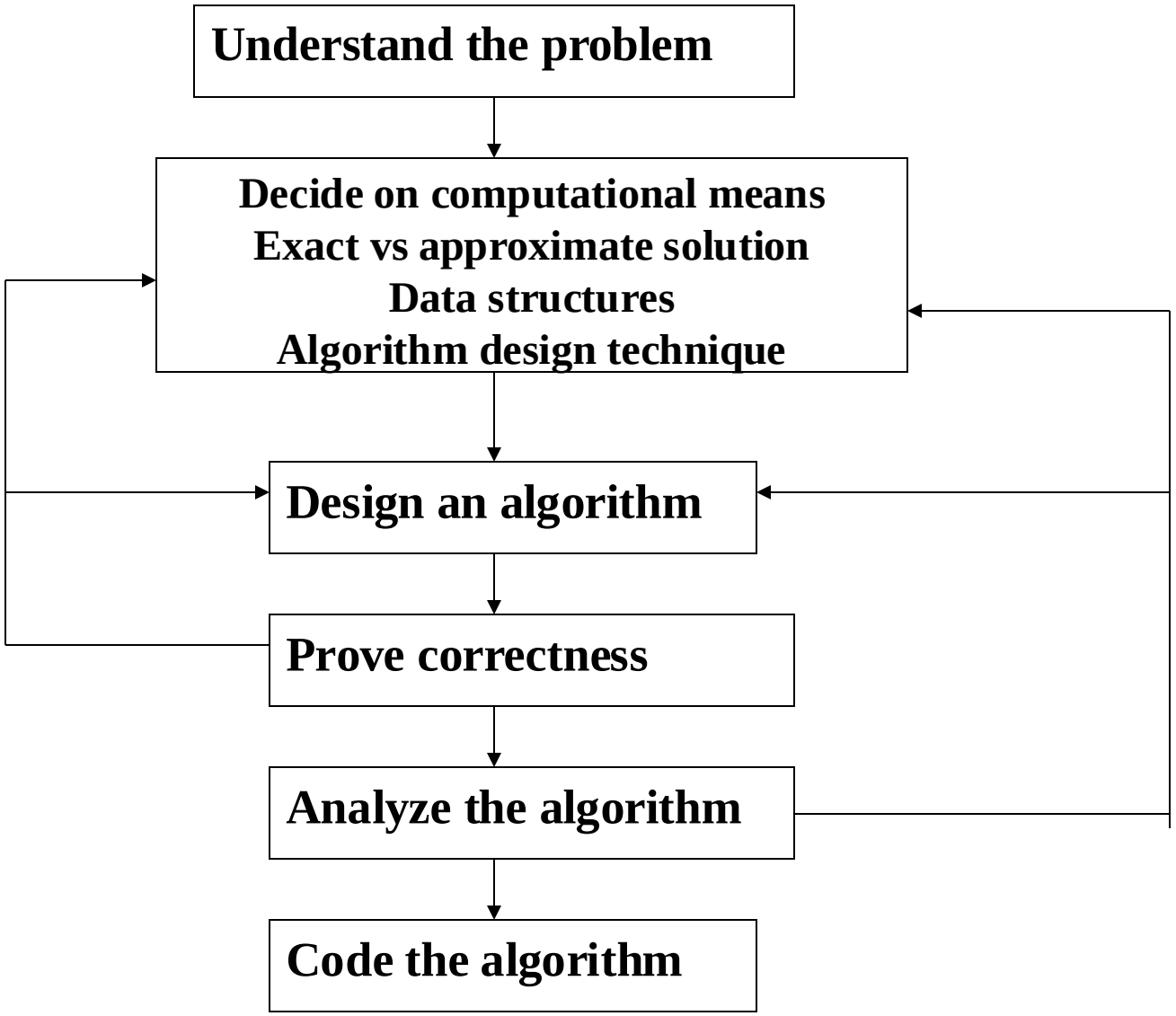


Recursion Tree for Permutations of String "ABC"

Algorithm

```
Algorithm Perm(a,k,n)
{
if(k=n)then write (a[1:n]);
else//a[k:n] has more than one permutation
//Generate the following
for i<- k to n do
{
t<- a[k];
a[k]<-a[i];
a[i]<-t;
Perm(a,k+1,n);
//All permutations of a[k+1:n]
t<- a[k];
a[k]<-a[i];
a[i]<-t;
}   }
```

Fundamentals of Algorithmic problem solving



We can consider algorithms to be procedural solutions to problems

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines.

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given.

Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms.

But often you will not find a readily available algorithm and will have to design your own.

The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value.

Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate Inputs

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture - random-access machine (RAM).

Its central assumption is that instructions are executed one after another, one operation at a time.
Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***

Sequential algorithm or serial **algorithm** is an **algorithm** that is executed **sequentially** – once through, from start to finish, without other processing executing – as opposed to concurrently or in parallel.

Parallel algorithm, as opposed to a traditional serial **algorithm**, is an **algorithm** which can do multiple operations in a given time.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately

In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***

There are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals.

Algorithm Design Techniques

An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

General techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work

Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task.

Some design techniques can be simply inapplicable to the problem in question.

Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.

Methods of Specifying an Algorithm

These are the two options that are most widely used nowadays for specifying algorithms.

Pseudocode is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language.

For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as for, if, and we use an arrow "←" for the assignment operation and two slashes "//" for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its ***correctness***.

That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.

A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs

Analyzing an Algorithm

We usually want our algorithms to possess several qualities.

After correctness, by far the most important is ***efficiency***.

In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.

Generality – Algorithm solves all problems of a certain type.

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs Programming an algorithm presents both a peril and an opportunity.
The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct.

They have developed special techniques for doing such proofs , but the power of these techniques of formal verification is limited so far to very small programs.

Analysis of Algorithms : The Need for Analysis

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem.

The need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as

A priori analysis − This is defined as theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors.

A posterior analysis − This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language. Next the chosen algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.

Algorithm analysis is dealt with
Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.
Time Complexity of an algorithm is the amount of computer time it needs to run to complete.

Space complexity

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

The space needed by the algorithm is given by

S(P)=C(fixed part)+ sp(Variable part)

fixed part

A fixed part that is independent of the characteristics (e.g., number, size)of the inputs and outputs.

This part typically includes the instruction space(i.e., space for the code), space for simple variables and fixed-size component variable space for constants, and soo.

variable part:

A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depend son instance characteristics) and the recursion stack space(insofar as this space depends on the instance characteristics.

Algorithm: 1

SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 - Stop

Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Algorithm: 2

```
Algorithm Max(A, n)
// A is an array of size n.
{
Result:=A[0];
for i :=1 to n do
if A[i] >Result then Result:=A[i];
return Result;
}
```

Variables i,n,Result – 1 unit each – fixed part
Variable A – n unit

- **if n value is 5, 5 units**
- **if n value is 100, 100 units**

Algorithm: 3

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

Here,

a-> 1
b-> 1
c-> 1

Total -> 3 Units

Algorithm: 4

```
AlgorithmSum(a,n)
{
S <- 0.0;
for i <- 1to n do
{
s:=s+ a[i];
}
return s;
}
```

i,n,s -> 1 unit each
a -> n units
Total -> n+1 units

Algorithm: 5

```
Algorithm RSum(a,n)
{
if(n <0) then
{
return0.0;
}
else
{
return RSum (a, n -l)+ a[n];
}
}
```

RSum(a,n) -> 1(a[n])+1(n)+1(return)

RSum(a,n-1) -> 1(a[n-1])+1(n)+1(return)

.

.

.

.

RSum(a,n-n) -> 1(a[n-n])+1(n)+1(return)

Total -> 3(n+1) units of memory required

Time Complexity

Time Complexity of an algorithm is the amount of computer time it needs to run to complete.

T(P) = compile time +execution Time
T(P) = tp (execution time)

Step Count:
Algorithm heading -> 0
For Braces -> 0
For expressions -> 1
For any looping statements -> the no. of times the loop is repeating

The time T(P)taken by a program P is the sum of the compile time and the run (or execution)time.
The compile time does not depend on the instance characteristics

we may assume that a compiled program will be run several times without recompilation C

This run time is denoted by tp (instance characteristics)

So,we could obtain an expression for tp(n) of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

Where n denotes the instance characteristics,

c_a, c_s, c_m, c_d, denotes the time needed for addition, subtraction, multiplication, division and so on…….

The number of steps any program statement is assigned depends on the kind of statement

For example, comments count as zero steps;
An assignment statement which does not involve any calls to other algorithms is counted as one step;

In an iterative statement such as the for, while, and repeat-until statements, 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 $i := \langle expr \rangle$ **to** $\langle expr1 \rangle$ **do**

while $(\langle expr \rangle)$ **do**

Each execution of the control part of a while statement is given a step count, equal to the number of step counts assignable to (expr).

The first execution of the control part of the for has a step count equal to the sum of the counts for (expr) and (exprl)

Remaining executions of the for statement have a step count of one;and soon.

```
Algorithm of abc(a,b,c)                          -> 0 Unit
{                                                -> 0 Units
return a + b*c+(a+ b-c)/(a+ b) + 4.0;            -> 1 Unit
}                                                -> 0 Unit
                                                 Total –> 1 Unit of time required
```

```
AlgorithmSum(a,n)          -> 0 Unit
{                          -> 0 Units
S <- 0.0;                  -> 1 Unit
for i <- 1to n do          -> n+1 Units
{                          -> 0 Units
s:=s+ a[i];                -> n units
return s;                  -> 1 unit
}                          -> 0 Unit
```

Total –> 2n+3 Unit of time required

```
Algorithm RSum(a,n)
{
if(n =0) then
{
return0.0;
}
else
{
return RSum (a, n -l)+ a[n];
}
}
```

$T(n)=2$ if $n=0$

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

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

$= 2+(2+T(n-2) = 2*2 + T(n-2)$

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

.

.

.

.

$= 2*n+T(n-n) = 2n+T(0)$

$T(n) = 2n+2$

Best-case efficiency

Best-case efficiency: Efficiency (number of times the basic operation will be executed) **for the best case input of size n.** ***i.e.***

The algorithm runs the fastest among all possible inputs of size n.

Best Case: In which we analyze the performance of an algorithm for the input, for which the algorithm takes less time or space.

Best-case Analysis Example: Linear Search

	0	1	2	3	4	5	6	7	8	9
Array	40	67	78	24	98	62	89	62	49	58

Best-case - Searching key element present at the first Index

Best Case Time = 1

B(1) = 1 = O(1)

B(n) = O(1)

Worst-case efficiency:

Algorithm efficiency depends on the **input size n.** And for some algorithms efficiency depends on **type of input**. We have best, worst & average case efficiencies.

Worst-case efficiency: Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.

Worst Case: In which we analyze the performance of an algorithm for the input, for which the algorithm takes long time or space.

Worst-case Analysis Example: Linear Search

	0	1	2	3	4	5	6	7	8	9
Array	40	67	78	24	98	62	89	62	49	58

Worst-case - Searching key element present at the last Index

Worst Case Time = n

$w(n) = n = O(n)$

$w(n) = O(n)$

Average-case efficiency

Average-case efficiency: Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.**

Average Case: In which we analyze the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

NOTE: NOT the average of worst and best case

Average-case efficiency Example: Linear Search

	0	1	2	3	4	5	6	7	8	9
Array	40	67	78	24	98	62	89	62	49	58

Average-case - All possible case time/No. of Cases

Average Time = $1+2+3+4+\ldots..+n/n = (n(n+1)/2)/n = (n+1)/2$

$A(n) = (n+1)/2 = O((n+1)/2)$

$A(n) = O((n+1)/2)$

In all these cases, the time complexities is based on the number of comparisons required

Asymptotic notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

Asymptotic notations

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis.

These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm

O − Big Oh
Ω − Big omega
θ − Big theta
o − Little Oh

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

fast

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

slow

High time efficiency

low time efficiency

$1<\log n<\sqrt{n}<n<n\log n<n^2<n^3 \ldots\ldots\ldots\ldots\ldots 2^n < 3^n <\ldots\ldots\ldots< n^n$

Big O Notation: The Big O notation defines an upper bound of an algorithm
Omegha Ω Notation : The Omegha Ω notation defines an lower bound of an algorithm
Big-Theta Θ notation : The Big-Thetha notation defines an Average bound of an algorithm

O (g(n)) = { f(n): there exist positive constants c and n0 such that f(n) <=c*g(n) for all n >= n0 }

Example 1:

f(n)=2n+3
2n+3<=10n
.'. f(n) = O(n)

Example 2:

f(n)=2n+3
2n+3<=7n
.'. f(n) = O(n)

Example 3:

$f(n)=2n^2+3n^2$
$2n^2+3n^2 <=5n^2$
.'. $f(n) = O(n^2)$

f(n)<=Cg(n)

All are true,

$f(n) = O(n)$
$f(n) = O(n^2)$
$f(n) = O(n^3)$
$f(n) = O(2^n)$

Example 1:

f(n)=2n+3
2n+3<=10n

If n=1,

= 2(1)+3<=10(1)

=2+3<=10 => 5<=10 ----------True

.'. f(n) = O(n)

Example 2:

f(n)=2n+3
2n+3<=7n

If n=1,

= 2(1)+3<=7(1)

=2+3<=7 => 5<=7 ----------True

.'. f(n) = O(n)

Example 3:

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

$2n^2+3n^2 <= 5n^2$

.'. $f(n) = O(n^2)$

<u>If n=1,</u>

$= 2(1)+3<=5(1^2)$

=2+3<=10 => 5<=10 ----------True

.'. f(n) = O(n)

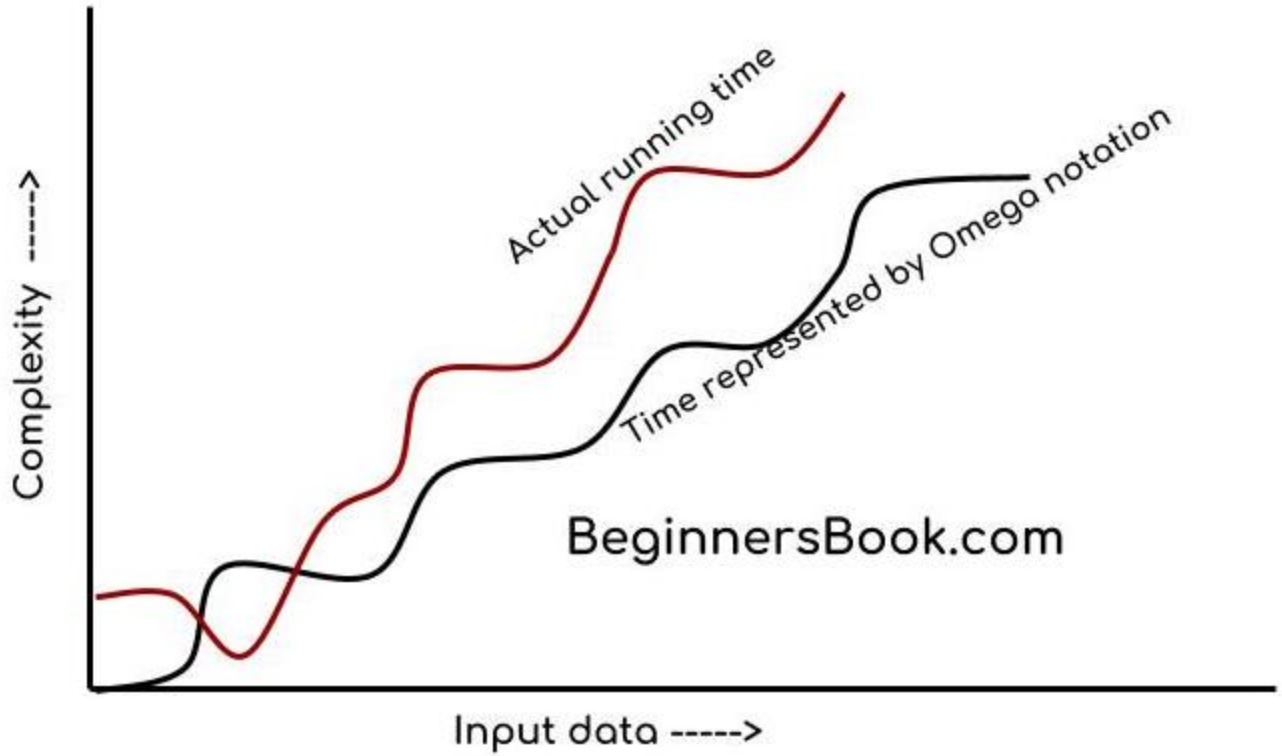
f(n)<=Cg(n)

All are true,

f(n) = O(n)
$f(n) = O(n^2)$
$f(n) = O(n^3)$
$f(n) = O(2^n)$

Omega notation : for Understanding

Omega notation specifically describes best case scenario. It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this**, it would at-least take the time represented by Omega notation or it can take more



$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 \ldots\ldots\ldots\ldots\ldots 2^n < 3^n < \ldots\ldots\ldots < n^n$

Big O Notation: The Big O notation defines an upper bound of an algorithm
Omegha Ω Notation : The Omegha Ω notation defines an lower bound of an algorithm
Big-Theta Θ notation : The Omegha Ω notation defines an Average bound of an algorithm

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that f(n) =>c*g(n) for all n >= n0 }

Example 1:

f(n)=2n+3
2n+3=>1n
.'. f(n) = Ω (n)

Example 2:

f(n)=2n+3
2n+3=>1*log n
.'. f(n) = Ω(n)

All are true,

f(n) = Ω(n)
f(n) = Ω(log n)

Example 1:

f(n)=2n+3
2n+3=>1n

If n=1,

= 2(1)+3=>1

=2+3=>1 => 5=>1 ----------True

.'. f(n) = Ω(n)

Example 2:

f(n)=2n+3

2n+3=>1*log n

=2+3=>1* log n ----------True

.'. f(n) = Ω(log n)

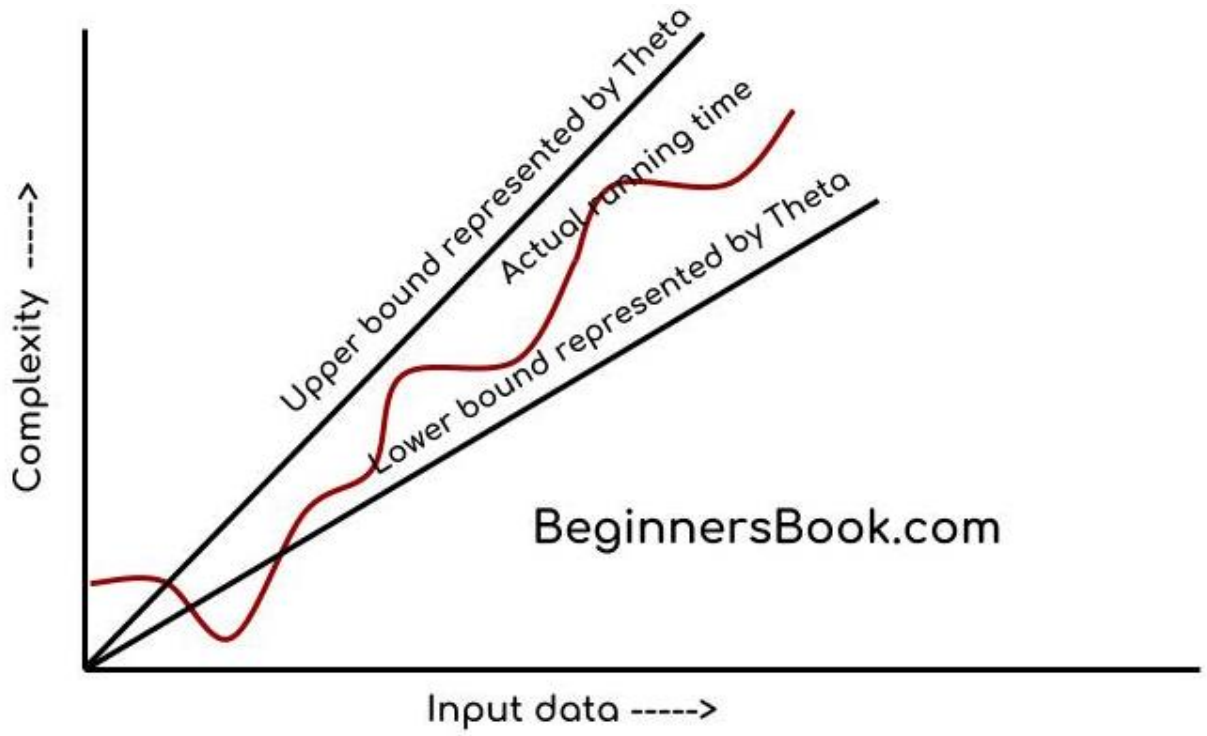
All are true,

f(n) = Ω(n)
f(n) = Ω(log n)

Theta notation

This notation describes both upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour.

In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.



$1<\log n<\sqrt{n}<n<n\log n<n^2<n^3 \ldots\ldots\ldots\ldots\ldots 2^n < 3^n <\ldots\ldots\ldots< n^n$

Big O Notation: The Big O notation defines an upper bound of an algorithm
Omegha Ω Notation : The Omegha Ω notation defines an lower bound of an algorithm
Big-Theta Θ notation : The Omegha Ω notation defines an Average bound of an algorithm

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $c_1 * g(n) <= f(n) <= c_2 * g(n)$ for all $n >= n0 \}$

Example 1:

$f(n)=2n+3$

$1*n<=2n+3<=5*n$

$c_1*g(n) <=f(n)<=c_2*g(n)$

$f(n) = \Theta(n)$ is true

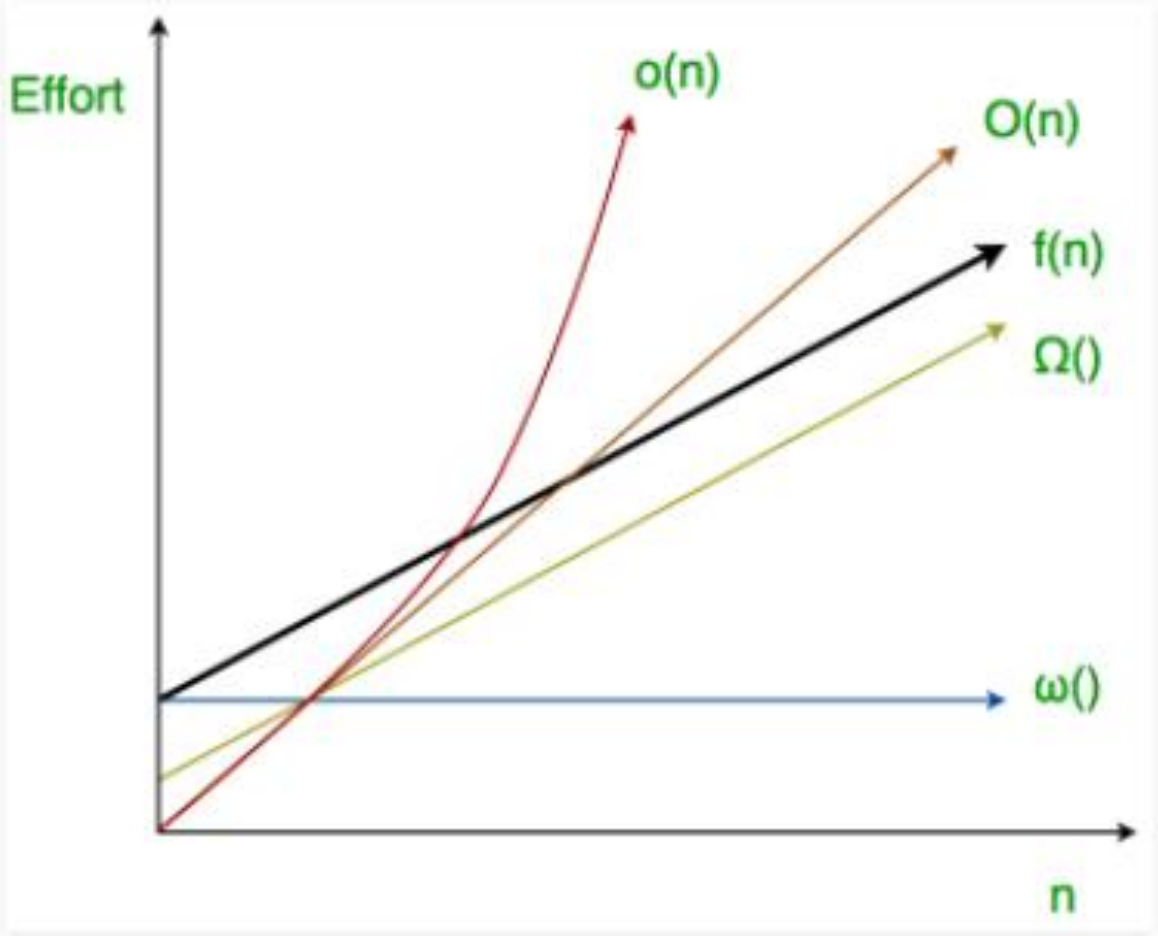
"Little-o" (o()) notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound.

"Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that $0 \leq f(n) < c*g(n)$.

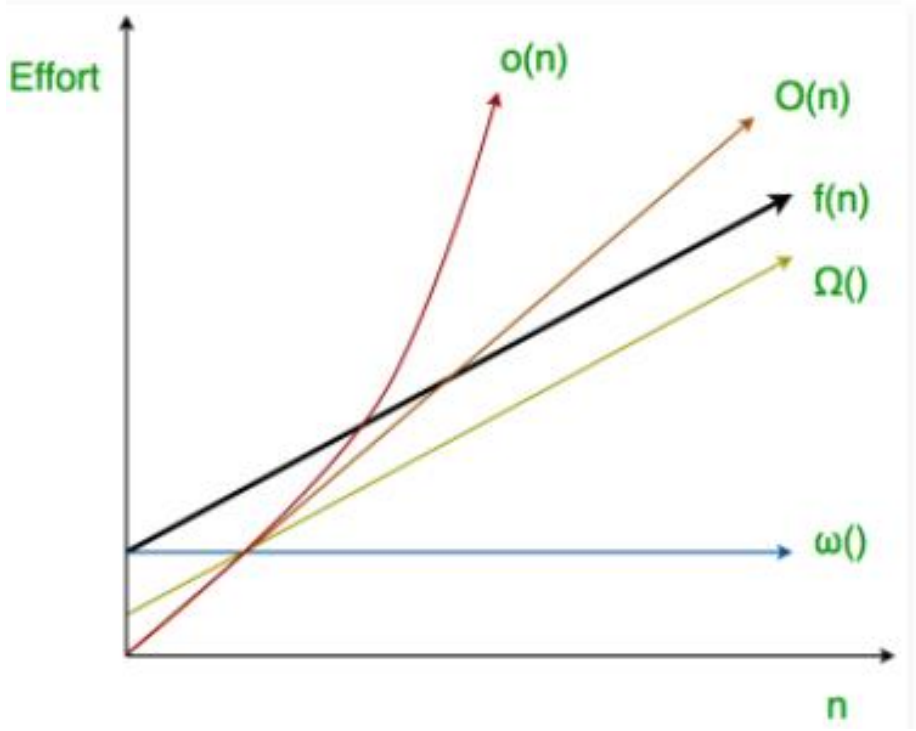
Thus, little o() means **loose upper-bound** of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.


Effort
o(n)
O(n)
f(n)
Ω()
ω()
n

Little o asymptotic notation

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).

In mathematical relation,
f(n) = o(g(n)) means
lim f(n)/g(n) = 0
n→∞



Summary

1.If **f(n) = Θ(g(n))**, then there exists positive constants c1, c2, n0 such that **0 ≤ c1.g(n) ≤ f(n) ≤ c2.g(n)**, for all n ≥ n0

2.If **f(n) = O(g(n))**, then there exists positive constants c, n0 such that **0 ≤ f(n) ≤ c.g(n)**, for all n ≥ n0

3.If **f(n) = Ω(g(n))**, then there exists positive constants c, n0 such that **0 ≤ c.g(n) ≤ f(n)**, for all n ≥ n0

4.If **f(n) = o(g(n))**, then there exists positive constants c, n0 such that **0 ≤ f(n) < c.g(n)**, for all n ≥ n0

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**

2. Identify algorithm's **basic operation**

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.

4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.

5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

ALOGORITHM 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 the largest element in A
currentMax <- A[0]
for i <- 1 to n − 1 do
if A[i] > currentMax
currentMax <- A[i]
return currentMax

- The obvious measure of an input's size here is the number of elements in the array, i.e., n.
- The operations that are going to be executed most often are in the algorithm's for loop.
- There are two operations in the loop's body:

  the comparison A[i] > maxval

  the assignment maxval <- A[i].

Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation.

- Let us denote C(n) the number of times this comparison is executed and try to find a formula expressing it as a function of size n.
- The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds between 1 and n —
- 1 (inclusively). Therefore, we get the following summation for C(n):

$$C(n) = \sum_{i=1}^{n-1} 1$$

- This is an easy sum to compute because it is nothing else but 1 repeated n — 1 times.

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \theta(n)$$

General Plan for Analyzing Efficiency of Non recursive Algorithms

- 1. Decide on a parameter (or parameters) indicating an input's size.
- 2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
- 3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
- 4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
- 5. Using standard formulas and rules of sum manipulation either find a closed-form formula for the count or, at the very least, establish its order of growth.

- Move constants out:

In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=1}^{u} c\,a_i = c \sum_{i=1}^{u} a_i \quad \text{----(R1)}$$

- Rearrange and recombine:

$$\sum_{i=1}^{u} (a_i \pm b_i) = \sum_{i=1}^{u} a_i \pm \sum_{i=1}^{u} b_i \quad \text{----(R2)}$$

and two summation formulas

- Basic sum formula:

$$\sum_{i=1}^{u} 1 = u - l + 1 \text{ where } l \leq u \text{ are some lower and upper integer limits --- (S1)}$$

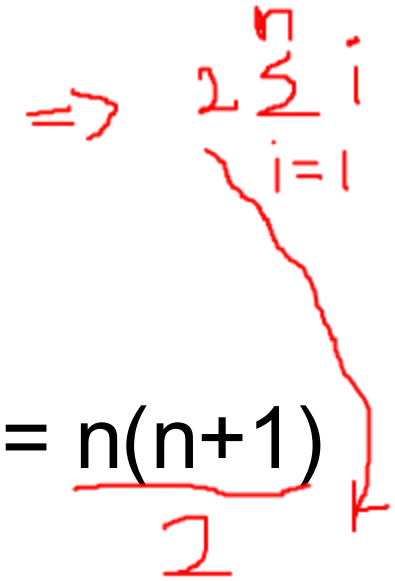
$$\sum_{i=0}^{n} i = 1+2+\ldots.+n = [n(n+1)]/2$$

- Gauss formula:[understanding]

1 2 3 4 ……n-1 n

\+

n n-1 ……..4 3 2 1



n+1 n+1……n+1 = n(n+1)/2

=> n(n+1)/2 => ½(n^2) => Θ($n^{2)}$

EXAMPLE 2 Consider the **element uniqueness problem:** check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

- **ALGORITHM** UniqueElements (A [0..n - 1])
- //Checks whether all the elements in a given array are distinct //Input:
- An array A[0..n - 1]
- //Output: Returns "true" if all the elements in A are distinct // and "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

- The natural input's size measure here is again the number of elements in the array, i.e., n.
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- Note, however, that the number of element comparisons will depend not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy.
- We will limit our investigation to the worst case only.

- By definition, the worst case input is an array for which the number of element comparisons Cworst(n) is the largest among all arrays of size n.

- An inspection of the innermost loop reveals that there ate two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely):

  -arrays with no equal elements

  -arrays in which the last two elements are the only pair of equal elements.

- For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits i + 1 and n - 1; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and n - 2. Accordingly, we get
- Summation for outer & Inner loop:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

=> (n-1)+(n-2)+(n-3).....+1

- => (n-1)+(n-2)+(n-3)……..+1
- => (n*(n-1))/2
- => $\Theta(n^{2})$

MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

We systematically apply the general framework to analyze the efficiency of recursive algorithms.

Example 1:

Compute the factorial function F(n) = n! for an arbitrary non negative integer n. Since, n! = 1 * 2 * ……. * (n-1) *n = n(n-1)!

For n ≥ 1 and 0! = 1 by definition, we can compute F(n) = F(n-1).n with the following recursive algorithm.

Factorial n! = n.n-1.n-2.n-3……..1

0! = 1

n! = n.(n-1)!

ALGORITHM F(n)

// Computes n! recursively

// Input: A nonnegative integer n

// Output: The value of n!

if n =0 return 1

else return F(n -1) * n

- For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion).
- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n).
- Since the function F(n) is computed according to the formula

$$F(n) = F(n-1) \cdot n \text{ for } n > 0,$$

The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

- $M(0) = 1$

- Indeed, M(n - 1) multiplications are spent to compute F(n - 1), and one more multiplication is needed to multiply the result by n.

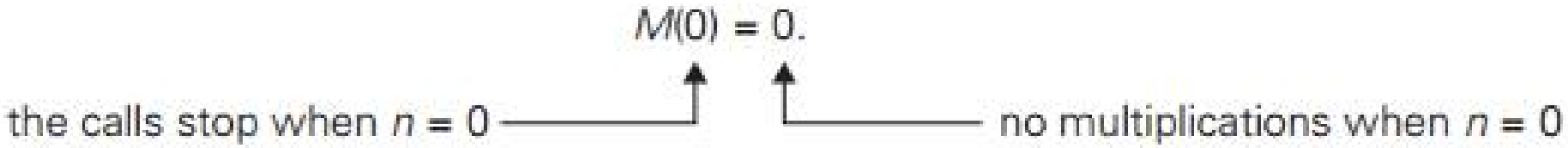
- Note that the equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n — 1. Such equations are called recurrence relations.
- Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics.
- Our goal now is to solve the recurrence relation

  M(n) = M(n - 1) + 1, i.e., to find an explicit formula for the sequence M(n) in terms of n only.
- That there is not one but infinitely many sequences that satisfy this recurrence. To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts

- We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if n = 0 return 1

This tells us two things. First, since the calls stop when ***n*** = 0**,** the smallest value of ***n*** for which this algorithm is executed and hence ***M(n)*** defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when ***n*** = 0**,** the algorithm performs no multiplications. Therefore, the initial condition we are after is



Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of multiplications M(n):

$$M(n) = M(n - 1) + 1 \text{ for } n > 0$$

$$M(0) = 0.$$

- We are dealing here with two recursively defined functions. The first is the factorial function F(n) itself; it is defined by the recurrence

$$F(n) = F(n - 1) * n \text{ for every } n > 0, F(0) = l.$$

- The second is the number of multiplications M(n) needed to compute F(n) by the recursive algorithm whose pseudocode was given at the beginning of the section.

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$

- After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line but also a general formula for the pattern:

$$M(n) = M(n-i) + i.$$

- Since it is specified for n = 0, we have to substitute i = n in the pattern's formula to get the ultimate result of our backward substitutions:
- M(n) = M(n - 1) + 1 = • • •
- = M(n - i) + i = - ----
- = M(n -n) + n
- = n.

General plan for investigating recursive algorithms.

- 1. Decide on a parameter (or parameters) indicating an input's size.
- 2. Identify the algorithm's basic operation.
- 3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
- 4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
- 5. Solve the recurrence or at least ascertain the order of growth of its solution.

EXAMPLE 2

The *Tower of Hanoi* puzzle.

In this puzzle, we (or mythical monks, if you do not like to move disks) have *n* disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution.

To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).

Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

Procedure:

Step 1:

If n=1, Move disc from source to destination

Step 2:

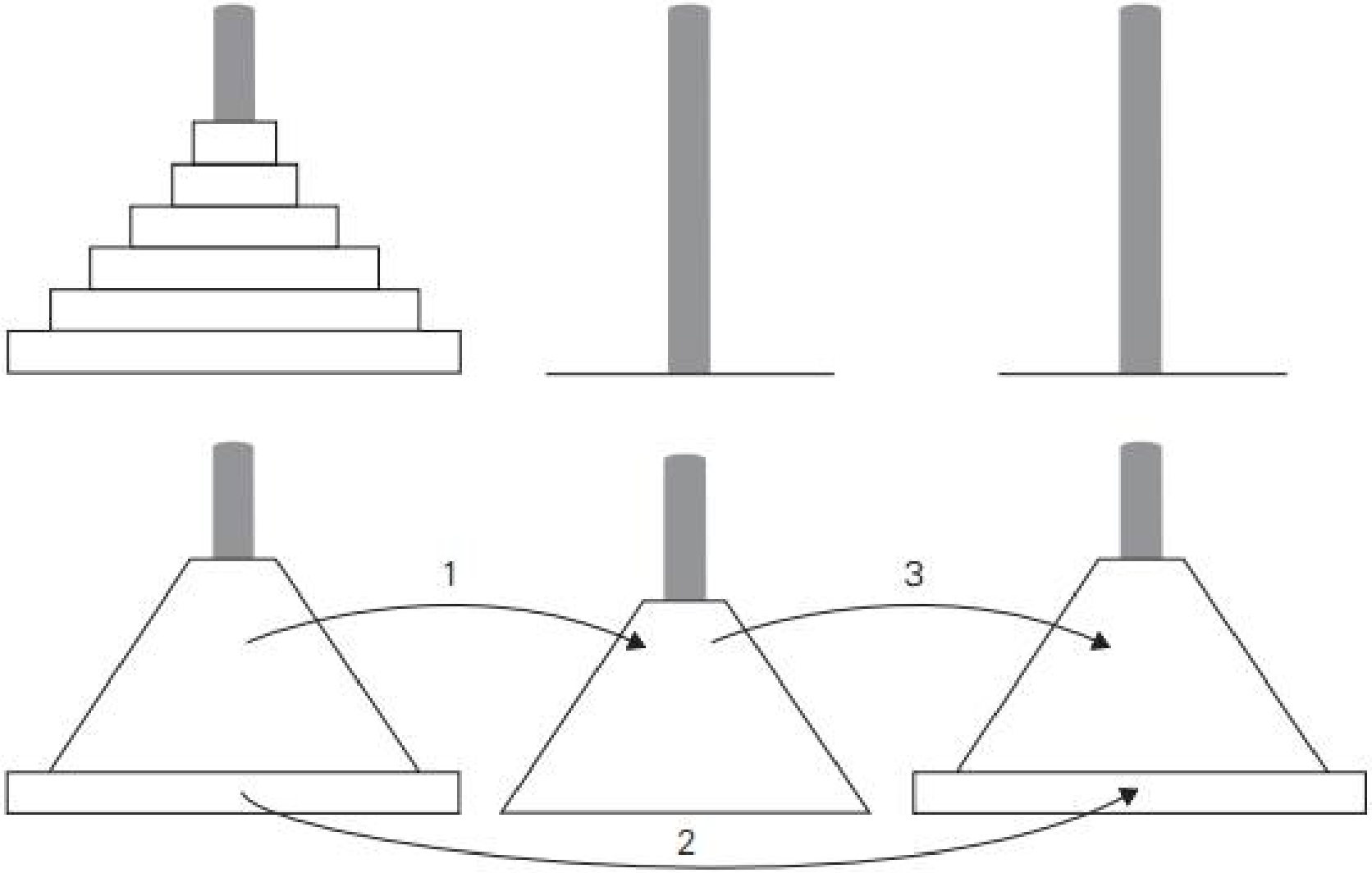
If n>1, Move n-1 disc to source to auxiliary using destination

Step 3:

Move disc from source to destination

Step 4:

Move n-1 disc from auxiliary to destination using source


1
3
2

Steps:

1. Problem size is *n*, the number of discs
2. The basic operation is moving a disc from rod to another
3. There is no worst or best case
4. Recursive relation for moving n discs

$$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1) = 2M(n\text{-}1) + 1$$

$$M(1) = 1$$

5. Solve using backward substitution.

$$M(n) = 2M(n\text{-}1) + 1$$

Steps:

1. Problem size is *n*, the number of discs
2. The basic operation is moving a disc from rod to another
3. There is no worst or best case
4. Recursive relation for moving n discs

$$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1) = 2M(n\text{-}1) + 1$$

$$M(1) = 1$$

5. Solve using backward substitution.

$$M(n) = 2M(n\text{-}1) + 1$$

- Apply the general plan outlined above to the Tower of Hanoi problem.

- The number of disks ***n*** is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation.

- Clearly, the number of moves ***M(n)*** depends on ***n*** only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \qquad \textbf{(2.3)}$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$M(n) = 2^{n-1} M(n - (n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

When a recursive algorithm makes more than a single call to itself, it can be useful for analysis purposes to construct a tree of its recursive calls.

In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls.

By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

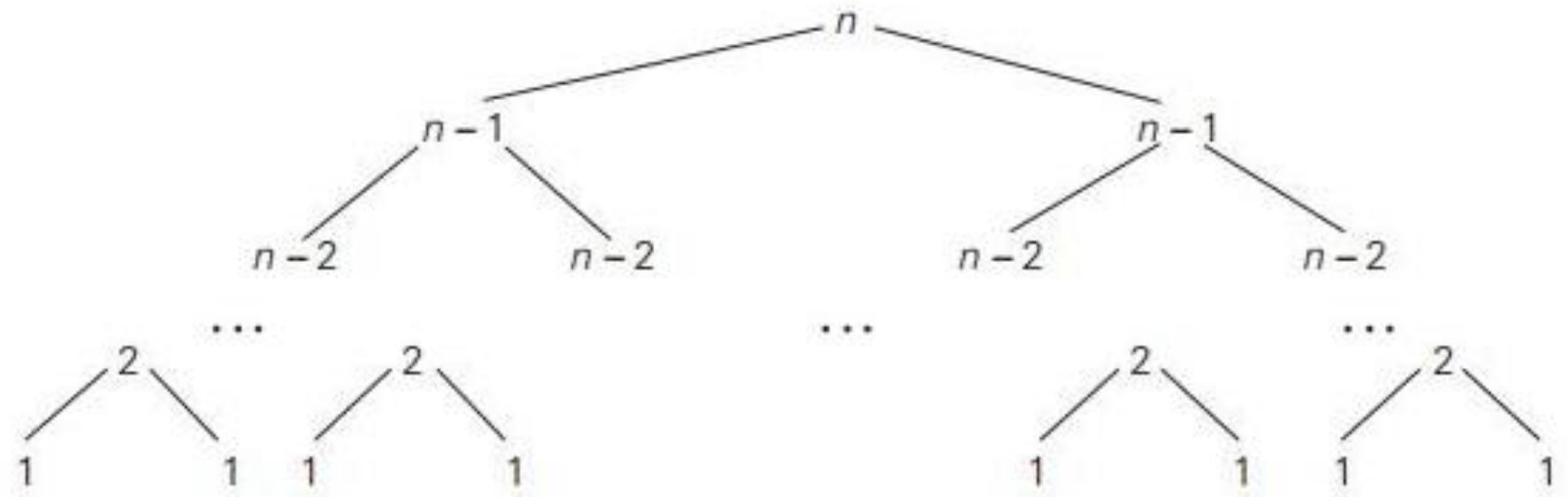
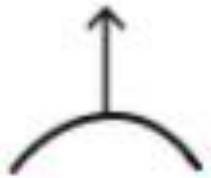


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

Fibonacci Series

Default



0 1 1 2 3 5

0 + 1 = 1

1 + 1 = 2

1 + 2 = 3

2 + 3 = 5

Example: Fibonacci numbers

In this section, we consider the Fibonacci numbers, a sequence of numbers as 0, 1, 1, 2, 3, 5, 8, …. That can be defined by the simple recurrence

$F(n) = F(n-1) + F(n-2)$ for $n > 1$

Two initial conditions,
$F(0) = 0, F(1) = 1$

The Fibonacci numbers were introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population.

Many more examples of Fibonacci-like numbers have since been discovered in the natural world, and they have even been used in predicting prices of stocks and commodities.

There are some interesting applications of the Fibonacci numbers in computer science as well.

For example, worst-case inputs for Euclid's algorithm happen to be consecutive elements of the Fibonacci sequence. Our discussion goals are quite limited here, however. First, we find an explicit formula for the nth Fibonacci number F(n).

Explicit Formula for the nth Fibonacci Number:

If we try to apply the method of backward substitutions to solve recurrence, we will fail to get an easily discernible pattern. Instead, let us take advantage of a theorem that describes solutions to a **homogeneous second-order linear recurrence with constant coefficients**

- $ax(n) + bx(n - 1) + cx(n - 2) = 0$, -----**(1)**

- where a, b, and c are some fixed real numbers ($a \neq 0$) called the coefficients of the recurrence and x(n) is an unknown sequence to be found.
- According to this theorem—recurrence (1) has an infinite number of solutions that can be obtained by one of the three formulas.
- Which of the three formulas applies for a particular case depends on the number of real roots of the quadratic equation with the same coefficients as recurrence(1).
- ar2 + br + c = 0**------ (2)**
- Quite logically, equation (2) is called the characteristic equation for recurrence (1).
- Let us apply this theorem to the case of the Fibonacci numbers.
- F(n) - F(n - 1) - F(n - 2) = 0. ---------(3)

- Its characteristic equation is r 2 ,

$$r - 1 = 0,$$

with the roots

$$r_{1,2} = (1 \pm \sqrt{1-4(-1)})/2 = (1 \pm \sqrt{5})/2$$

Algorithms for Computing Fibonacci Numbers

- Actually, the sequence grows so fast that it is the size of the numbers rather than a time-efficient method for computing them that should be of primary concern here.

- Also, for the sake of simplicity, we consider such operations as additions and multiplications at unit cost in the algorithms that follow.

- Since the Fibonacci numbers grow infinitely large (and grow rapidly)

ALGORITHM F(n)

- //Computes the nth Fibonacci number recursively by using its definition //Input:
- A non negative integer n
- //Output: The nth Fibonacci number
- **if n < 1 return n**
- **else return F(n - 1) + F(n - 2)**

Analysis:

- The algorithm's basic operation is clearly addition, so let

A(n) be the number of additions performed by the algorithm in computing F(n).

- Then the numbers of additions needed for computing F(n — 1) and F(n — 2) are A(n — 1) and A(n — 2), and the algorithm needs one more addition to compute their sum.

Thus, we get the following recurrence for A(n):

$A(n) = A(n - 1) + A(n - 2) + 1$ for $n > 1$.

$A(0)=0$, $A(1) = 0$.

- The recurrence A(n) — A(n — 1) — A(n — 2) = 1 is quite similar to recurrence (3) but its right-hand side is not equal to zero.
- Such recurrences are called **inhomogeneous recurrences.**
- There are general techniques for solving inhomogeneous recurrences, but for this particular recurrence, a special trick leads to a faster solution.

We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

$[A(n) + 1] - [A(n -1) + 1]- [A(n - 2) + 1] = 0$ and substituting

$B(n) = A(n) + 1$:

$B(n) - B(n - 1) - B(n - 2) = 0$

$B(0) = 1, B(1) = 1.$

- This homogeneous recurrence can be solved exactly in the same manner as recurrence was solved to find an explicit formula for F(n).

- We can obtain a much faster algorithm by simply computing the successive elements of the Fibonacci sequence iteratively, as is done in the following algorithm..

- **ALGORITHM** Fib(n)
- //Computes the nth Fibonacci number iteratively by using its definition
- //Input: A nonnegative integer n
- //Output: The nth Fibonacci number
- F[0]<-0; F[1]<-1
- for i <- 2 to n do
- F[i]«-F[i-1]+F[i-2]
- **return** F[n]

- This algorithm clearly makes n - 1 additions. Hence, it is linear as a function of n and "only“ exponential as a function of the number of bits b in n's binary representation.
- Note that using an extra array for storing all the preceding elements of the Fibonacci sequence can be avoided:
- Storing just two values is necessary to accomplish the task

- The third alternative for computing the nth Fibonacci number lies in using a formula. The efficiency of the algorithm will obviously be determined by the efficiency of an exponentiation algorithm used for computing $ø^n$.
- If it is done by simply multiplying ø by itself n - 1 times, the
- algorithm will be in $\theta(n) = \theta(2b)$.
- There are faster algorithms for the exponentiation problem.
- Note also that special care should be exercised in implementing this approach to computing the nth Fibonacci number.
- Since all its intermediate results are irrational numbers, we would have to make sure that their approximations in the computer are accurate enough so that the final round-off yields a correct result.

- Finally, there exists a θ (log n) algorithm for computing the nth Fibonacci number that manipulates only integers. It is based on the equality

- F(n-1) F(n) 0 1^n
- F(n) F (n + 1) 1 1 *for n ≥ 1*
- and an efficient way of computing matrix powers.

Example 2: the algorithm to find the number of binary digits in the binary representation of a positive decimal integer.

- The algorithm for finding the binary representation of a positive decimal integer in English is as follows:
- 1. Let the decimal number be N.
- 2. Let the B be a string variable to hold the binary representation of the N.
- 3. Repeat the following steps 4 to 6 until N becomes 0.
- 4. Divide the decimal number N by 2. Let the remainder be R and quotient be Q.
- 5. Add the remainder R to B (from right to left) to be the next digit of the binary number.
- 6. Assign the quotient Q as the new decimal number N

For example,

quotient	remainder	interpretation (not part of algorithm)
241		
120	1	$241 = 120*2 + 1$
60	0	$120 = 60*2 + 0$
30	0	$60 = 30*2 + 0$
15	0	$30 = 15*2 + 0$
7	1	$15 = 7*2 + 1$
3	1	$7 = 3*2 + 1$
1	1	$3 = 1*2 + 1$
0	1	$1 = 0*2 + 1$
0	0	$0 = 0*2 + 0$
0	0	
:	:	

$$241_{ten} = 11110001_{two}.$$

- **ALGORITHM** BinRec(n) : -
- //Input: A
- positive decimal integer n
- //Output: The number of binary digits in n's binary representation **if** n = 1 **return 1**
- **else return** BinRec(n/2) + 1

- Let us set up a recurrence and an initial condition for the number of additions A(n) made by the algorithm.

- The number of additions made in computing BinRec(n/2) is A(n/2), plus one more addition is made by the algorithm to increase the returned value by 1.

- This leads to the recurrence
- A(n) = A(n/2) + 1 for n > 1.

- Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is A(1) = 0

- The presence of [n/2] in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2.
- Therefore, the standard approach to solving such a recurrence is to solve it only for n — 2k and then take advantage of the theorem called the **smoothness rule** which claims that under very broad assumptions the order of growth observed for n = 2k gives a correct answer about the order of growth for all values of n.

- (Alternatively, after getting a solution for powers of 2, we can sometimes fine tune this solution to get a formula valid for an arbitrary n.)

- So let us apply this recipe to our recurrence, which for
- n = 2k takes the form

$$A(2^k) = A(2^{k-1}) + 1 \text{ for } k > 0, \qquad A(2^0) = 0$$

Now backward substitutions encounter no problems:

$$A(2^k) = A(2^{k-1}) + 1 \qquad \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1$$

$$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 \text{ substitute } A(2^{k-2})$$

$$= A(2^{k-3}) + 1 = [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$$

$$= \quad \ldots\ldots\ldots\ldots\ldots$$

$$= \quad A(2^{k-i}) + i$$

$$= \quad \ldots\ldots\ldots\ldots\ldots$$

$$= \quad A(2^{k-k}) + k$$

Thus, we end up with

$$A(2^k) = A(1) + k = k$$

or, after returning to the original variable $n = 2^k$ and, hence, $k = \log_2 n$,

$$A(n) = \log_2 n \in \theta(\log n).$$

Important Problem Types

In this section, we are going to introduce the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Sorting:

- The ***sorting problem*** is to rearrange the items of a given list in non decreasing order. Of course, for a problem to be meaningful, the nature of the list items must allow such an ordering.
- As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings.
- Records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees
- In the case of records, we need to choose a piece of information to guide sorting.
- For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***.

Why would we want a sorted list?

To begin with, a sorted list can be a required output of a task such as ranking Internet search results or ranking students by their GPA scores.

Further, sorting makes many questions about the list easier to answer.

The most important of them is searching: it is why dictionaries, telephone books, class lists, and so on are sorted.

Sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms and data compression.

On the one hand, there are a few good sorting algorithms that sort an arbitrary array of size n using about $n \log_2 n$ comparisons

Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations.

Some of the algorithms are simple but relatively slow, while others are faster but more complex; some work better on randomly ordered inputs, while others do better on almost-sorted lists; some are suitable only for lists residing in the fast memory, while others can be adapted for sorting large files stored on a disk; and so on.

Two properties of sorting algorithms

A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where i < j, then in the sorted list they have to be in positions i and j.

We have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.

Generally speaking, algorithms that can exchange keys located far apart are not stable, but they usually work faster;

The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be ***in-place*** if it does not require extra memory, except, possibly, for a few memory units.

There are important sorting algorithms that are in-place and those that are not.

Searching

The searching problem deals with finding a given value, called a search key, in a given set

There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching.

The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

For searching, too, there is no single algorithm that fits all situations best.

Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on

Unlike with sorting algorithms, there is no stability problem, but different issues arise.

In applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: an addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation

Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-world applications.

String Processing

A ***string*** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}.

One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it string matching Several algorithms that exploit the special nature of this type of searching have been invented.

Graph Problems

A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.

Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games.

Basic graph algorithms include graph-traversal algorithms, shortest-path algorithms, and topological sorting for graphs with directed edges.

Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.

The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through *n* cities that visits every city exactly once.

In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering.

The ***graph-coloring problem*** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color.

This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

The traveling salesman problem and the graph coloring problem are examples of combinatorial problems. These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints. A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost.

Combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint. Their difficulty stems from the following facts.

First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances.

Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

Moreover, most computer scientists believe that such algorithms do not exist. This conjecture has been neither proved nor disproved, and it remains the most important unresolved issue in theoretical computer science.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons.

The ancient Greeks were very much interested in developing procedures for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles,circles, and so on.

Here is an algorithms for only two classic problems of computational geometry: the closest-pair problem and the convex-hull problem.

The closest-pair Problem is self-explanatory: given n points in the plane, find the closest pair among them. The convex-hull problem asks to find the smallest convex polygon that would include all the points of a given set.

Numerical Problems

Numerical problems another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.

The majority of such mathematical problems can be solved only approximately.

Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately.

Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications.

Fundamental Data Structures

A ***data Structure*** can be defined as a particular scheme of organizing related data items.

Linear Data Structures

The two most important elementary data structures are the array and the linked list. A(one-dimensional) ***array*** is a sequence of *n* items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's ***index.***

In the majority of cases, the index is an integer either between 0 and $n - 1$ or between 1 and *n*. Some computer languages allow an array index to range between any two integer bounds *low* and *high*, and some even permit non numerical indices to specify, for example, data items corresponding to the 12 months of the year by the month names.

Arrays are used for implementing a variety of other data structures. Prominent among them is the ***string*** a sequence of characters from an alphabet terminated by a special character indicating the string's end. Strings composed of zeros and ones are called ***binary strings*** or ***bit strings***.

A ***linked list*** is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called ***pointers*** to other nodes of the linked list.

In a ***singly linked list*** each node except the last one contains a single pointer to the next element

To access a particular node of a linked list, one starts with the list's first node and traverses the pointer chain until the particular node is reached. Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located.

On the positive side, linked lists do not require any preliminary reservation of the computer memory, and insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers

To start a linked list with a special node called the ***header***. This node may contain information about the linked list itself, such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

Another extension is the structure called the ***doubly linked list*** in which every node, except the first and the last, contains pointers to both its successor and its Predecessor.

A ***list*** is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element.

A ***stack*** is a list in which insertions and deletions can be done only at the end. This end is called the ***top*** because a stack is usually visualized not horizontally but Vertically.

As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in a "last-in–first-out" (LIFO) fashion—exactly like a stack of plates if we can add or remove a plate only from the top.

A ***queue***, on the other hand, is a list from which elements are deleted from one end of the structure, called the ***front*** (this operation is called ***dequeue***), and new elements are added to the other end, called the ***rear*** (this operation is called ***enqueue***).

Consequently, a queue operates in a "first-in–first-out" (FIFO) fashion

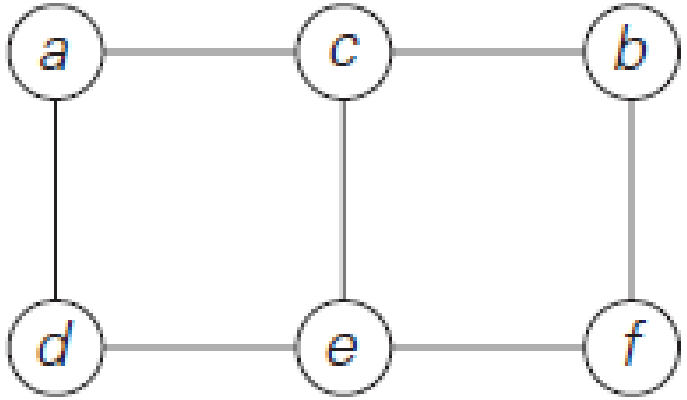
Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a priority queue.

A ***priority queue*** is a collection of data items from a totally ordered universe The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.

Graphs

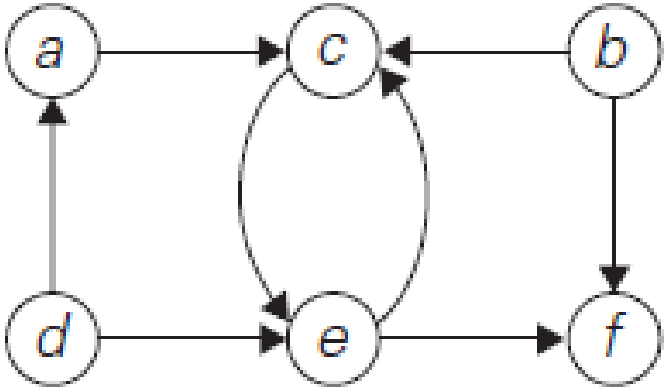
A graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs." Formally, a ***graph****G* = *V,E* is defined by a pair of two sets: a finite nonempty set *V* of items called ***vertices*** and a set *E* of pairs of these items called ***edges***
A graph *G* is called ***undirected*** if every edge in it is undirected.

A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***.



Directed Graph

V = {*a, b, c, d, e, f* },
E = {*(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)*}.



Undirected Graph

V = {*a, b, c, d, e, f* },
E = {*(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)*}.

Graph Representations Graphs for computer algorithms are usually represented in one of two ways:

Adjacency matrix
Adjacency lists

The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge.

The of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

Adjacency matrix

a	→	c	→	d			
b	→	c	→	f			
c	→	a	→	b	→	e	
d	→	a	→	e			
e	→	c	→	d	→	f	
f	→	b	→	e			

Adjacency lists

A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***.

An interest in such graphs is motivated by numerous real-world applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem.

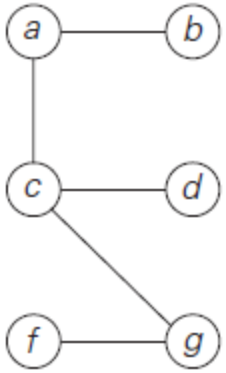
A ***path*** from vertex *u* to vertex *v* of a graph *G* can be defined as a sequence of adjacent (connected by an edge) vertices that starts with *u* and ends with *v*. If all vertices of a path are distinct, the path is said to be ***simple.***

A ***cycle*** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, *f* , *h*, *i*, *g*, *f* is a cycle in the graph in Figure 1.9. A graph with no cycles is said to be ***acyclic***.

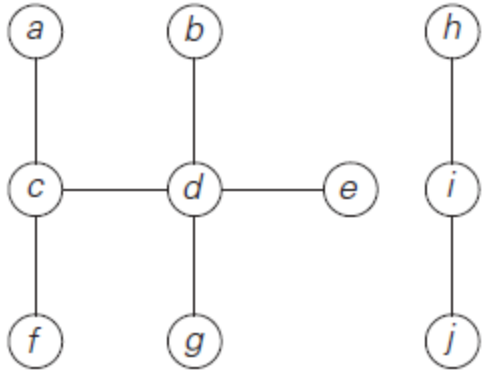
Trees

A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph .

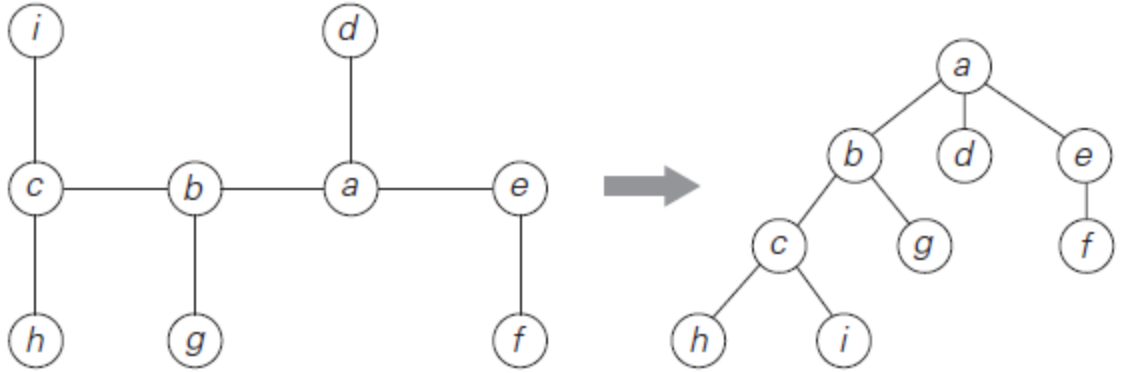
A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree



(a) Tree.



(b) Forest



(a) Free tree. (b) Its transformation into a rooted tree.

An ***ordered tree*** is a rooted tree in which all the children of each vertex are ordered.

A ***binary tree*** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a ***left child*** or a ***right child*** of its parent; a binary tree may also be empty

Sets and Dictionaries

A ***set*** can be described as an unordered collection (possibly empty) of distinct items called ***elements*** of the set.

The first considers only sets that are subsets of some large set *U*, called the ***universal set***. If set *U* has *n* elements, then any subset *S* of *U* can be represented by a bit string of size *n*, called a ***bit vector***.

PROBLEMS

Find the O notation of the following function.

$f(n)=3n+2$

$0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0$

Here $f(n)=3n+2$ $g(n)=n$ $c=4$

$3n+2 \leq 4n$ for all $n \geq n_0$

$3n+2 \leq 4n$

If n = 1, LHS = 5, RHS = 4, False

If n = 2, LHS = 8, RHS = 8, True

If n = 3, LHS = 11, RHS = 12, True

If n = 4, LHS = 14, RHS = 16, True

The above equation is True, when $n \geq 2$, $n_0 = 2$

$f(n) = 3n + 2$

$0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$

Here $f(n) = 3n + 2$ $g(n) = n$ $c = 4$ $n_0 = 2$

$3n + 2 \leq 4\ n$ for all $n \geq 2$

Therefore $f(n) = \mathbf{O(g(n))}$

Find the O notation of the following function

$f(n) = 4n^3 + 2n + 3$

$0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$

Here $f(n) = 4n^3 + 2n + 3$ $g(n) = n^3$ $c=5$

$4n^3 + 2n + 3 \leq 5\ n^3$ for all $n \geq n_0$

$$4n^3 + 2n + 3 \leq 5\ n^3$$

If n=1, LHS=9 ,	RHS=5,	False
If n=2, LHS=39 ,	RHS=40,	True
If n=3, LHS=117 ,	RHS=135,	True
If n=4, LHS=267 ,	RHS=320,	True

The above equation is True when $n \geq 2$

Therefore $n_0=2$

$f(n) = 4n^3 + 2n + 3$

$0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$

Here $f(n) = 4n^3 + 2n + 3$ $g(n) = n^3$ $c=5$ $n_0=2$

$4n^3 + 2n + 3 \leq 5\ n^3$ for all $n \geq 2$

Therefore $f(n) = \mathbf{O(g(n))}$

$4n^3 + 2n + 3 = \mathbf{O(n^3)}$

$4n^3+2n+3 \leq 5n^3$

If n = 1, LHS = 9, RHS = 5, False

If n = 2, LHS = 39, RHS = 40, True

If n = 3, LHS = 117, RHS = 135, True

If n = 4, LHS = 267, RHS = 320, True

The above equation is True, when $n \geq 2$, $n_0 = 2$

Find the **Ω** notation of the following function.

$f(n) >= c*g(n),\ n \geq n_0$

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

$3n^2+n >= c.g\ (n)$

$3n^2+n >= 3n^2 = \mathbf{\Omega}(n^2)$

Where $n <= n_0$ and $n=1$

$3n^2+n>=3n^2$

If n = 1, LHS = 3, RHS = 3, True

If n = 2, LHS = 14, RHS = 12, True

If n = 3, LHS = 30, RHS = 27, True

If n = 4, LHS = 52, RHS = 48, True

The above equation is True, when n>=1, $n_0 = 1$

Find the **Ω** notation of the following function.

$f(n) >= c*g(n),\ n \geq n_0$

$f(n)=2n^2+n$

$2n^2+n >= c.g\ (n)$

$2n^2+n >= 2n^2 = \mathbf{\Omega}(n^2)$

Where $n <= n_0$ and $n=1$

$2n^2+n>=2n^2$

If n = 1, LHS = 2, RHS = 2, True

If n = 2, LHS = 10, RHS = 8, True

If n = 3, LHS = 21, RHS = 18, True

If n = 4, LHS = 36, RHS = 32, True

The above equation is True, when n>=1, n_0 =1

Find the Θ notation of the following function

$f(n) = 3n + 2$

$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$

$c_1\, g(n) \leq f(n)$ for all $n \geq n_0$

Here $f(n) = 3n + 2$ $g(n) = n$ $c_1 = 3$ $n_0 = 1$

$3n \leq 3n + 2$ for all $n \geq 1$

$f(n) \leq c_2\, g(n)$ for all $n \geq n_0$

Here $f(n) = 3n + 2$ $g(n) = n$ $c_2 = 4$ $n_0 = 2$

$3n + 2 \leq 4n$ for all $n \geq 2$

$3n \leq 3n + 2 \leq 4n$ for all $n \geq 2$ $c_1 = 3$ $c_2 = 4$ $n_0 = 2$

Therefore $f(n) = \Theta(g(n))$ ➔ **$3n + 2 = \Theta(n)$**

1.Solve the recurrence relation T (n) = 1 if n=1 , 2T (n-1) if n>1

$T(n) = 2T(n-1)$

$= 2[2T(n-2)] = 2^2T(n-2)$

$= 4[2T(n-3)] = 2^3T(n-3)$

$= 8[2T(n-4)] = 2^4T(n-4)$…… (Eq.1)

Repeat the procedure for i times

$T(n) = 2^i T(n-i)$

Put n-i=1 or i= n-1 in…… (Eq.1)

$T(n) = 2^{n-1} T(1)$

$= 2^{n-1} .1$ {T (1) =1 .....given}

$= 2^{n-1}$

$T(n) = 2T(n-1)$

$T(n-1) = 2T(n-2)$

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

$T(n-3) = 2T(n-4)$

2. Solve the recurrence relation T (n) = T (n-1) +1 and T (1) = θ (1).

T (n) = T (n-1) +1
= (T (n-2) +1) +1 = (T (n-3) +1) +1+1
= T (n-4) +4 = T (n-5) +1+4
= T (n-5) +5
= T (n-k) + k

Where k = n-1

T(n-(n-1)+n-1

T(n-n+1)+(n-1)

T (1) = θ (1)

T (n) = θ (1) + (n-1)

= 1+n-1
= n = θ (n).

T (n) = T (n-1) +1

T (n-1) = T (n-2) +1

T (n-2) = T (n-3) +1

T (n-3) = T (n-4) +1

Solve the recurrence relation

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

$T(n) = T(n-1)+n$…….(1)

$T(n) = [T(n-2)+n-1]+n$

$T(n) = T(n-2)+(n-1)+n$ ……(2)

$T(n) = [T(n-3)+n-2]+(n-1)+n$

$T(n) = T(n-3)+(n-2)+(n-1)+n$…….(3)

Continue for k times,

$T(n) = T(n-k)+(n-(k-1))+(n-(k-2)+n)$……..+(n-1)+n

Assume n-k=0 ; n=k

$T(n) = T(n-n)+(n-n+1)+(n-n+2)+$……..(n-1)+n

$T(n) = T(0)+1+2+3$…….(n-1)+n

$T(n) = 1 + (n(n+1)/2) = \Theta (n^2)$

$T(n) = T(n-1)+n$

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

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

$T(n-3) = T(n-4)+n-3$

Solve the recurrence relation

$X(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$

a) $X(n) = X(n-1) + 5$, for $n > 1$, $X(1) = 0$

$$
\begin{aligned}
x(n) &= x(n-1) + 5 \\
&= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\
&= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\
&= \dots \\
&= x(n-i) + 5 \cdot i \\
&= \dots \\
&= x(1) + 5 \cdot (n-1) = 5(n-1).
\end{aligned}
$$

$$= \Theta(n)$$

Useful Property Involving the Asymptotic Notations

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non negative integer n_1 such that $t_1(n) \le c_1 g_1(n)$ for all $n \ge n_1$.

Similarly, since $t_2(n) \in O(g_2(n))$, $t_2(n) \le c_2 g_2(n)$ for all $n \ge n_2$.

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

$$\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)]$$

$$\leq c_3 2 \max\{g_1(n), g_2(n)\}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

Assignment: 1

1. Solve the recurrence relation:

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

MCQ's REVISION

The recurrence relation capturing the optimal time of the Tower of Hanoi problem with n discs is $M(n) = 2M(n\text{-}1) + 1$

Performance measurement is concerned with obtaining the space and time complexity is Space and time Complexity

A **permutation** is a mathematical technique that determines the number of possible arrangements in a set when the order of the arrangements matters.

For 5 disks, how many moves are required in towers of Hanoi Problem ?

For 4 disks, how many moves are required in towers of Hanoi Problem ?

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

Space Complexity – 3 Units

```
AlgorithmSum(a,n)
{
S <- 0.0;
for i <- 1to n do
{
s:=s+ a[i];
}
return s;
}
```

Space complexity - n+1

Time Complexity – 2n+3

```
Algorithm RSum(a,n)
{
if(n <0) then
{
return0.0;
}
else
{
return RSum (a, n -l)+ a[n];
}
}
```

Space complexity - 3(n+1)

Time Complexity – 2n+2

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

Time Complexity – 1 Units

Which characteristics represents “Each step must be precisely defined; Each instruction is clear and unambiguous” – Definiteness

The ***sorting problem*** is to rearrange the items of a given list in non decreasing order.

An algorithm is a sequence of unambiguous instructions for solving a problem

For 2 disks, how many moves are required in towers of Hanoi Problem

For 1 disks, how many moves are required in towers of Hanoi Problem

f(n)<=c.g(n) represents Big-O notation

f(n) =>c*g(n) represents Ω notation

c_1*g(n) <=f(n)<=c_2*g(n) represents Θ notation

What is the Best case analysis of linear search ? O(1)

Cg(n) is an asymptotic lower bound of f(n) represents Omega notation
Cg(n) is an asymptotic upper bound of f(n) represents Big-O notation

Brute Force Method is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved

An algorithm is said to be **recursive** if the same algorithm is invoked in the body (direct recursive).

Algorithm ***A*** is said to be **indirect recursive** if it calls another algorithm which in turn calls A.

The Average case notation can be represented by

Average case efficiency of Linear search is A(n) = O((n+1)/2)

Worst case efficiency of Linear search is w(n)- O(n)

For 3 disks, how many moves are required in towers of Hanoi Problem

Sequential algorithm or serial **algorithm** is an **algorithm** that is executed **sequentially** – once through, from start to finish, without other processing executing – as opposed to concurrently or in parallel.

we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***

Parallel algorithm, as opposed to a traditional serial **algorithm**, is an **algorithm** which can do multiple operations in a given time.

Pseudocode is a mixture of a natural language and programming language like constructs

There are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.

4.If **f(n) = o(g(n))**, then there exists positive constants c, n0 such that **0 ≤ f(n) < c.g(n)**, for all n ≥ n0, represents little o notation.

Average case complexity of the **element uniqueness problem - $\Theta(n^{2})$**

The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through *n* cities that visits every city exactly once

The ***graph-coloring problem*** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color.

The traveling salesman problem and the graph coloring problem are examples of combinatorial problems

Numerical problems another large special area of applications, are problems that involve mathematical objects of continuous nature

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

fast

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

slow

High time efficiency

low time efficiency

A ***data Structure*** can be defined as a particular scheme of organizing related data items.

A ***priority queue*** is a collection of data items from a totally ordered universe The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.

graph*G = V,E* is defined by a pair of two sets: a finite nonempty set *V* of items called ***vertices*** and a set *E* of pairs of these items called ***edges.*** A graph *G* is called ***undirected*** if every edge in it is undirected.

The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge.

A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***.

A ***path*** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.

A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree

A ***binary tree*** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a ***left child*** or a ***right child*** of its parent; a binary tree may also be empty

A ***set*** can be described as an unordered collection (possibly empty) of distinct items called ***elements*** of the set.