Data Structures – 4CS203

Dr. N. L. Gavankar

What is Data Structure ?

- Data Structure (DS) is one of the most fundamental subject in CSE.
- In depth understanding of this topic is very important especially when you are into development/programming domain where you build efficient software system and application

What is Data Structure ?

- **Definition**
- A Data Structure is a data organization, management and storage format that enables efficient access and modification.
- A data Structure is an aggregation of atomic and composite data into a set with a defined relationship. In this definition structure means a set of rules that holds the data together.

What is Data Structure ?

In simple words

DS is a way in which data is stored on a computer.

Why do we need Data Structure ?

DS is a way of storing and organizing information in a computer so that it can be retrieved and used most productively.

- Each data structure allows data to be stored in specific manner
- DS allows efficient data search and retrieval
- Specific DS are decided to work a specific problem
- It allows to manage large amount of data

Why do we need Data Structure ?

Examples

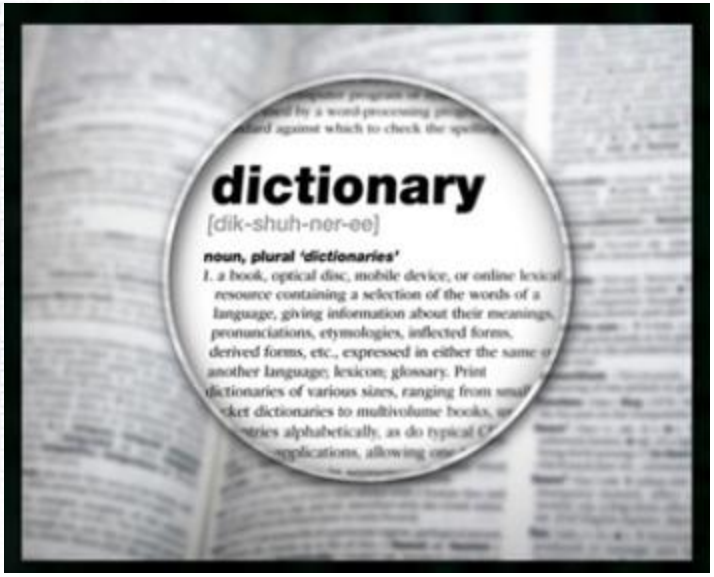


Table1

ID	FirstName	Surname	Age
1	John	Jones	35
2	Tracey	Smith	25
3	Anne	McNeil	30
4	Andrew	Francis	37
5	Gillian	Carpenter	32
6	Karen	Rogers	22
7	Amy	Sanders	42
8	Kevin	White	38
9	Charlie	Anderson	40
10	Mary	Brown	26
11	Andrew	Smith	32
12	James	Francis	28
13	Karen	Jones	30
14	Edward	Kent	32
15	Jenny	Smith	26
16	Angela	Jones	41
(New)			

What is a Program

Steps:

- Get a problem statement
- Think about a logic
- Write an algorithm
- Design a flowchart
- Write a code and execute
- Validate the output and submit

Pseudocode

- One of the most common tools for defining **algorithms**.
- **Pseudocode** is an English-like representation of the algorithm logic.
- **Describes** what must be done without showing unnecessary details, such as error messages.

Pseudocode Example

```
Algorithm sample (pageNumber)
This algorithm reads a file and prints a report.
   Pre    pageNumber passed by reference
   Post   Report Printed
          pageNumber contains number of pages in report
   Return Number of lines printed
 1 loop (not end of file)
   1   read file
   2   if (full page)
       1   increment page number
       2   write page heading
   3   end if
   4   write report line
   5   increment line count
 2 end loop
 3 return line count
end sample
```

Pseudocode Example

- **Header**
  - Name, List of parameters, Precondition, Post condition.
- **Statement Number, Purpose, Conditions, and Return**
- **Variables –** Intelligent data name
  - Do not use Single Char. and generic names.
- **Statement Construct**
  - Sequence, Selection, and Loop.

Analysis of Algorithm - Not every line of code is explained, only those points that either need to be emphasized or that may require some clarification. Also include Efficiency consideration

The Abstract Data Type

- **Programming Evolution**

Spaghetti : Non Structured Linear Programs

Modular Programming : Programs organized in functions

Structured Programming : structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines (Formulated in 1970 by Edsger Dijkstra and Niklaus Wirth)

The Abstract Data Type

- **Atomic and Composite Data**

Atomic Data : Data that consist of a single piece of information; i.e they can not be divided into other meaningful pieces of data. Example integer 34523.

Composite Data : Opposite of Atomic Data. Composite data can be broken out into subfields that have meaning. Example PRN No. 2018BTECV00088

The Abstract Data Type

- **Data Type**

A Data Type consists of two parts: a set of data and the operations that can be performed on the data. Thus, the integer type consists of values and operations.

1. A set of Values
2. A set of operations on values

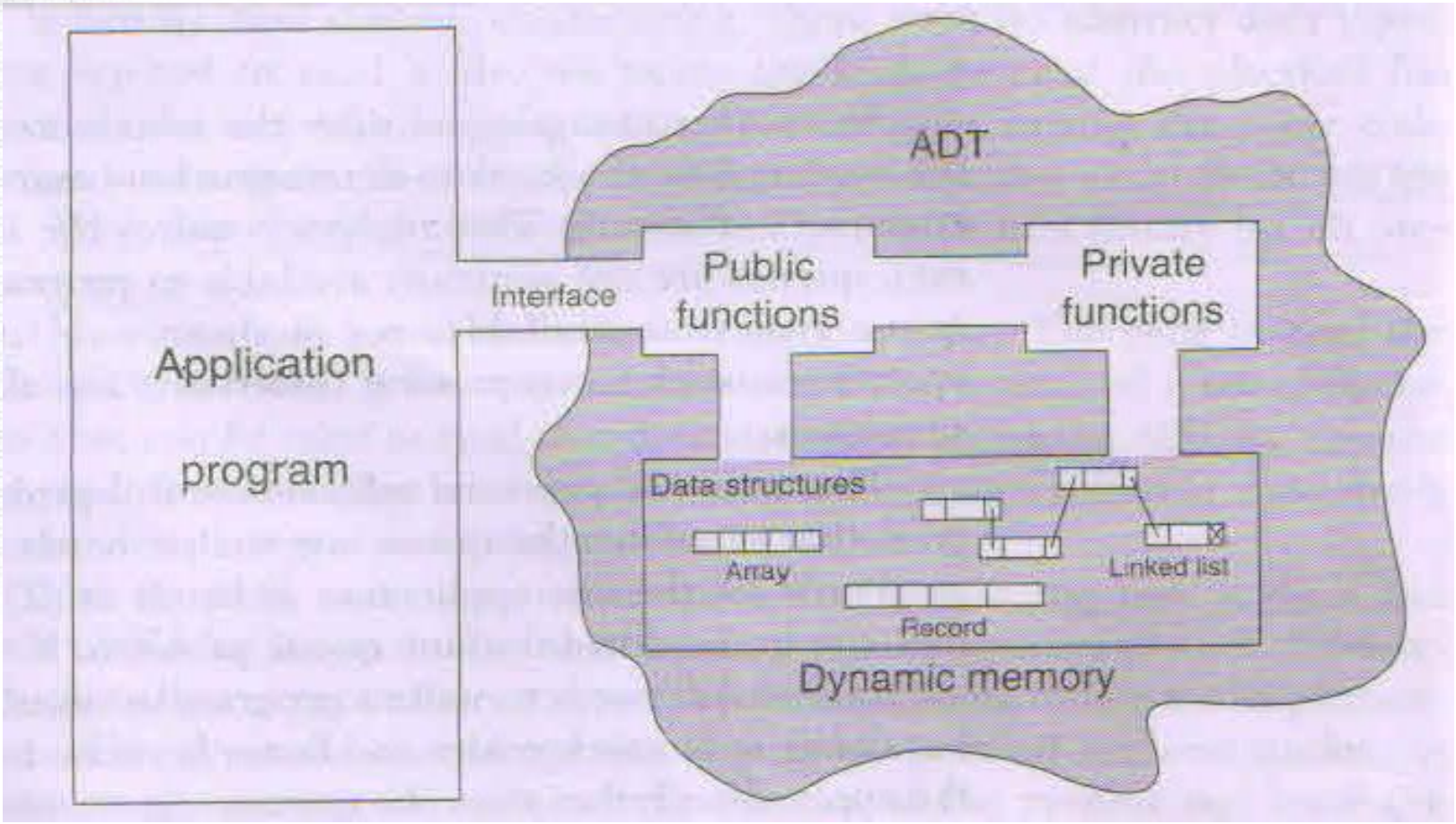
The Abstract Data Type

- **ADT**
- Reusability of code
- With ADT users are not concerned with how the task is done but rather with what it can do
- ADT consists of set of definitions that allow programmers to use the functions while hiding the implementation details.
- The generalization of operations with unspecified implementation is known as abstraction
- We abstract the essence of the process and leave the implementation details hidden

The Abstract Data Type

- **ADT Definition**
- ADT is a data declaration packaged together with operations that are meaningful for the data type
- We encapsulate the data and the operations on the data and then we hide them from the user.

Model for an Abstract Data Type



Algorithm Efficiency

- **Algorithmics** is the systematic study of the fundamental techniques used to design and analyze efficient algorithms

  -- Brassard and Bratley

- If the function is linear i.e. if it contains no loop or recursions the efficiency is function of the number of instructions it contains. In this case efficiency depends on speed of computer.

Algorithm Efficiency

- **We** generally discuss the algorithm's efficiency as a function of the number of elements to be processed.
- The general format is

$$f(n) = efficiency$$

The study of algorithm efficiency focuses on loop.

Algorithm Efficiency

- **Linear Loops :** How many times the body of the loop is repeated.

```
for (i = 0; i < 1000; i++)
    application code
```

Efficiency is directly proportional to the number of iterations

$$f(n) = n$$

Algorithm Efficiency

- **Linear Loops :** How many times the body of the loop is repeated.

```
for (i = 0; i < 1000; i += 2)
    application code
```

$$f(n) = n/2$$

Algorithm Efficiency

- **Logarithmic Loops :** The controlling variable is multiplied or divided in each iteration.

Multiply Loops

```
for (i = 1; i <= 1000; i *= 2)
      application code
```

Divide Loops

```
for (i = 1000; i >= 1; i /= 2)
      application code
```

Algorithm Efficiency

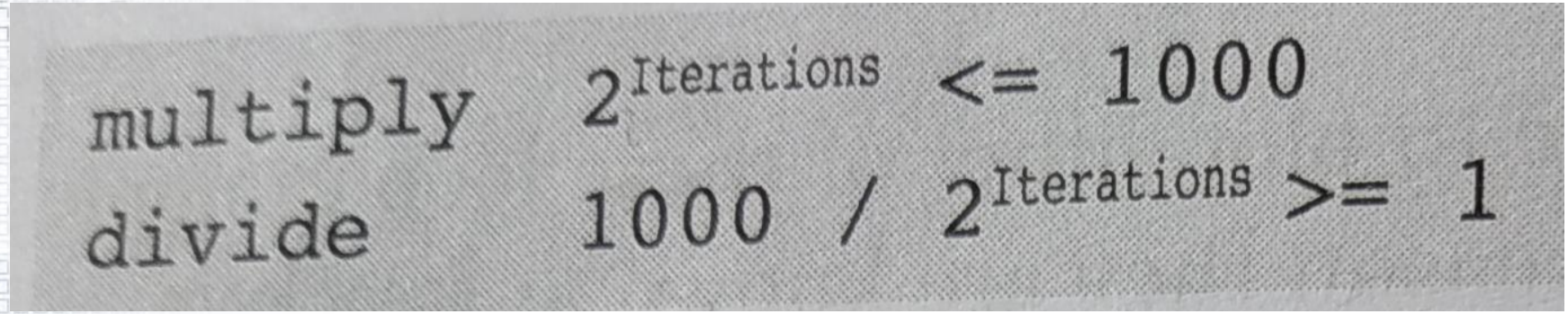
- **Logarithmic Loops :** The controlling variable is multiplied or divided in each iteration.

Multiply		Divide	
Iteration	Value of i	Iteration	Value of i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

3 Analysis of Multiply and Divi

Algorithm Efficiency

- **Logarithmic Loops :** The controlling variable is multiplied or divided in each iteration.



- Generalizing analysis

$$f(n) = \log n$$

Algorithm Efficiency

- **Nested Loops** loop that contain loop
- When we analyze nested loops, we must determine how many iterations each loop completes.
- The total is then the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

Iterations = outer loop iterations * inner loop iterations

Algorithm Efficiency

- **Nested Loops** loop that contain loop

```
for (i = 0 ; i < 10 ; i ++)
    {
        for ( j = 0 ; j < 10 ; j ++)
         {
            Print  " Hi"
          }
     }
```

Algorithm Efficiency – Nested Loops

```
for (i = 0; i < 10; i++)
    for (j = 1; j <= 10; j *= 2)
        application code
```

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        application code
```

```
for (i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
        application code
```

Algorithm Efficiency – Nested Loops

- **Linear Logarithmic**
- **Quadratic**
- **Dependent Quadratic**

Algorithm Efficiency – Nested Loops

- **Linear Logarithmic**

```
for (i = 0; i < 10; i++)
    for (j = 1; j <= 10; j *= 2)
        application code
```

Inner loop *log 10*

Inner loop is controlled by outer loop

= 10 log 10

Generalized as *f (n) = n log n*

Algorithm Efficiency – Nested Loops

- **Quadratic loop –** The number of times the inner loop executes is the same as the outer loop.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        application code
```

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

Algorithm Efficiency – Nested Loops

- **Dependent Quadratic** – the number of iterations of the inner loop depends on the outer loop.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
        application code
```

- The number of iterations in the body of the inner loop is calculated as
- 1 + 2 +3 + . . . + 9 + 10 = 55

Algorithm Efficiency – Nested Loops

- If we compute the average of this loop it is 55/10 = 5.5
- Same as the iteration (10) plus 1 divided by 2.
- Mathematically *(n + 1) / 2*
- Multiplying the inner loop by the number of times the outer loop is executed gives as

$$f(n) = n\left(\frac{n+1}{2}\right)$$

Big-O Notation

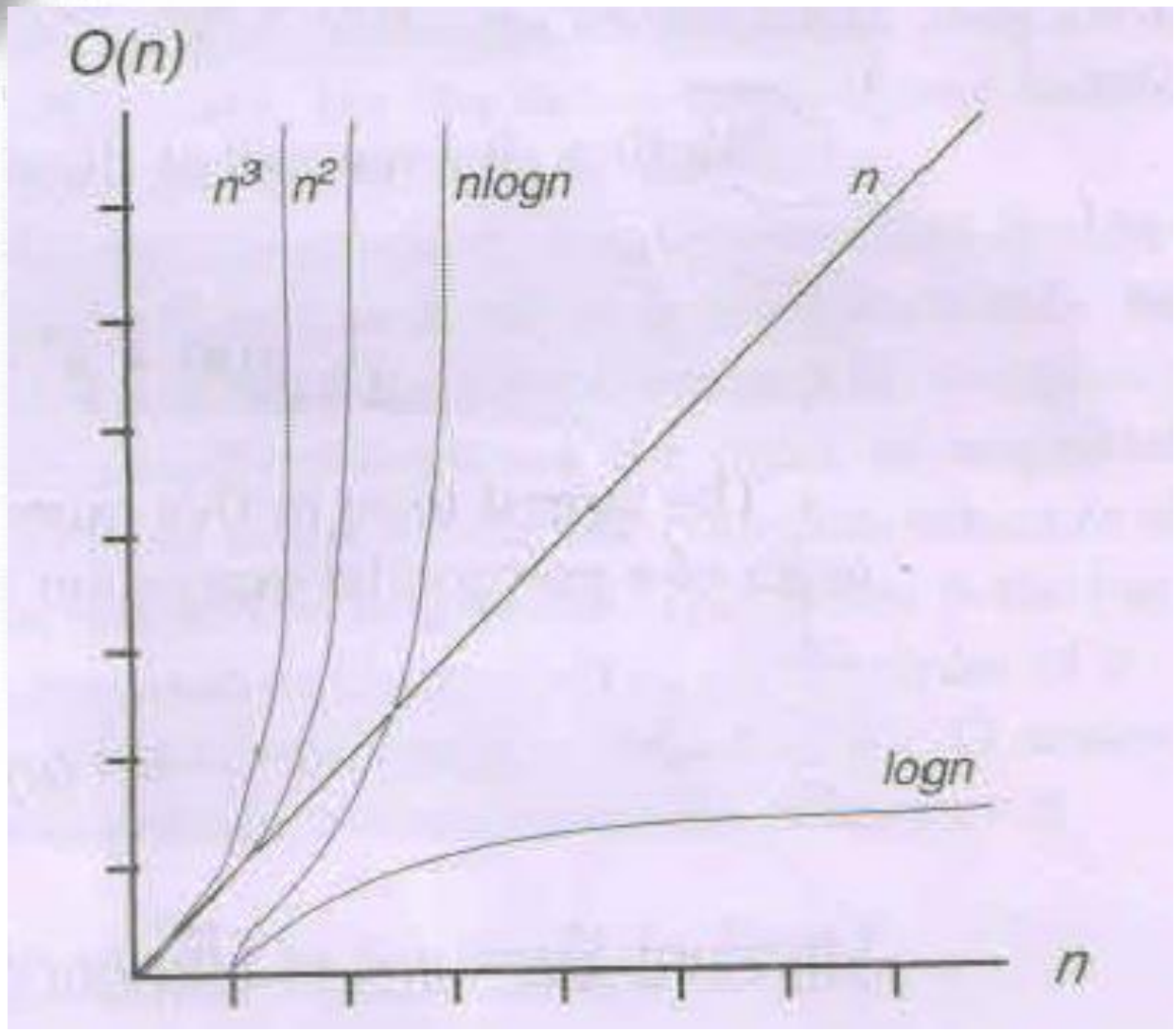
- *f(n) : Function of number of elements*
- Dominant factor usually determines the order of magnitude of result.
- Therefore, we don't need to determine the complete measure of efficiency, only the factor that determines the magnitude.
- This factor is the Big-O and expressed as *O (n) i.e Order of n*

Algorithm Efficiency – Nested Loops

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

Measures of Efficiency for $n = 10{,}000$

Algorithm Efficiency – Nested Loops



Recursion

- Recursion is a repetitive process in which an algorithm/function calls itself

A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.

Recursion – Example - Factorial

Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
   Pre  n  is the number to be raised factorially
   Post n! is returned
1  set i to 1
2  set factN to 1
3  loop (i <= n)
   1   set factN to factN * i
   2   increment i
4  end loop
5  return factN
end iterativeFactorial
```

Recursion – Example - Factorial

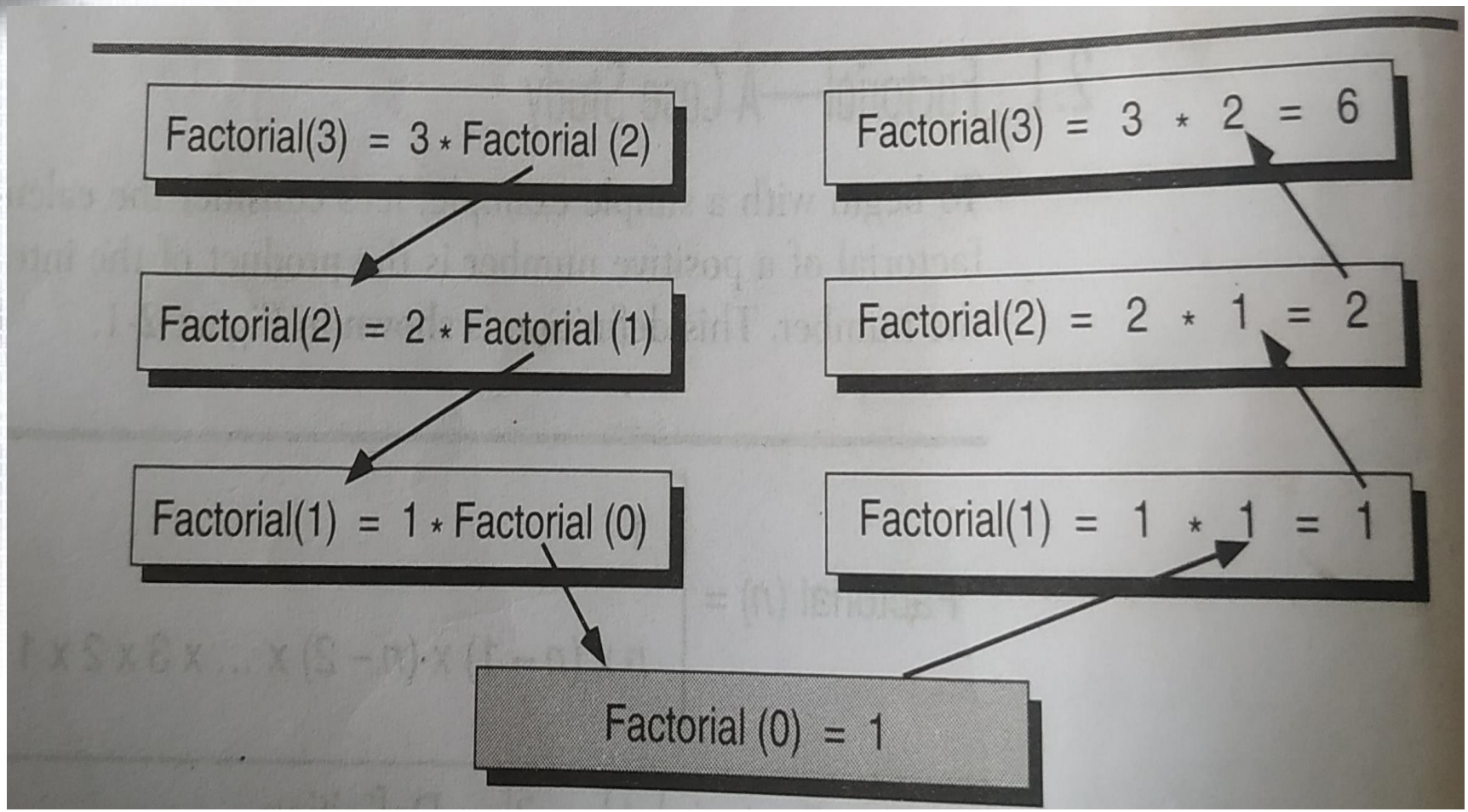
Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
   Pre    n  is the number being raised factorially
   Post   n! is returned
1  if (n equals 0)
   1   return 1
2  else
   1   return (n * recursiveFactorial (n - 1))
3  end if
end recursiveFactorial
```

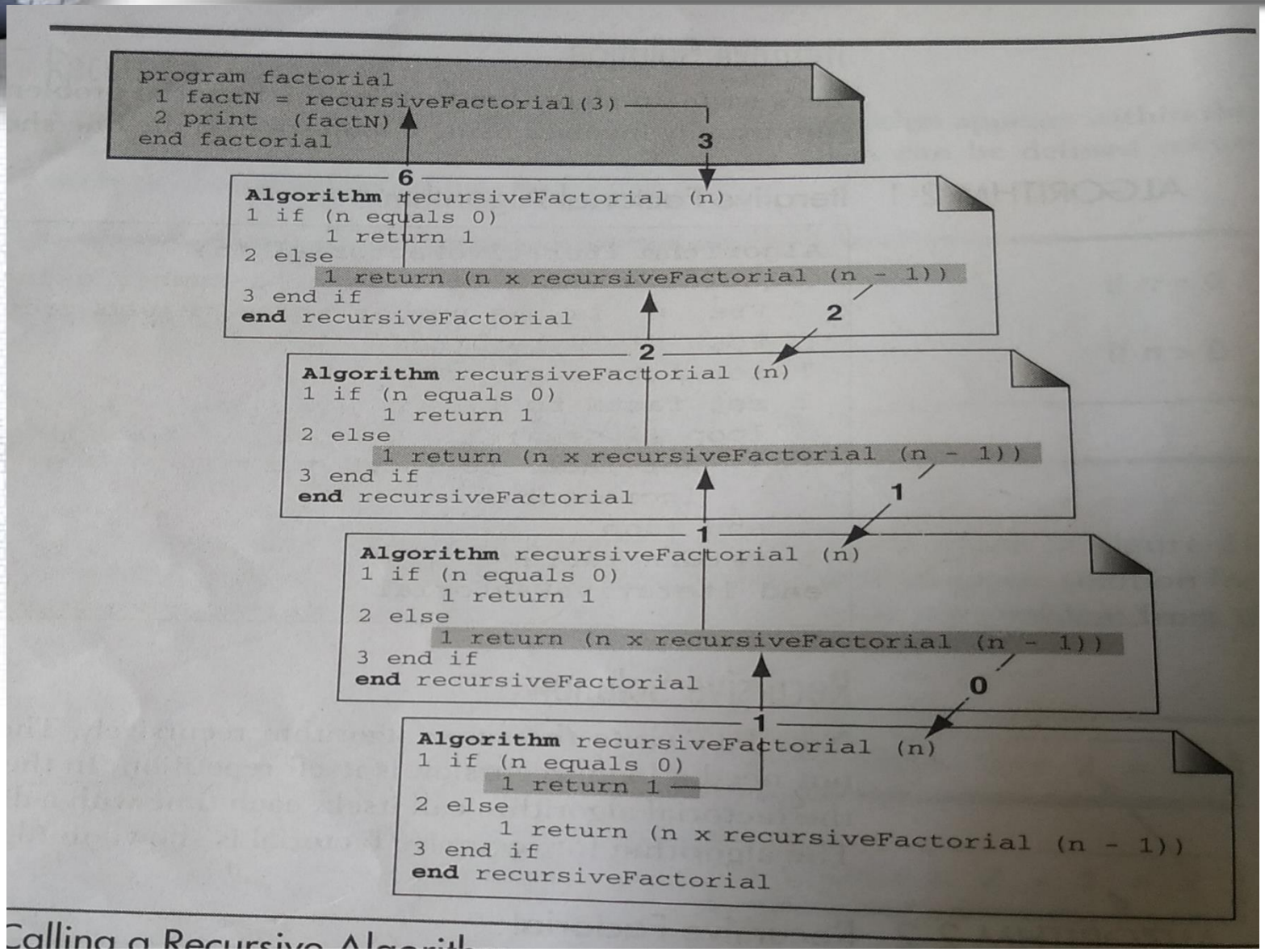
Recursion – Example - Factorial

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Recursion – Example - Factorial



Recursion – Example - Factorial



Calling a Recursive Algorith

Recursion

- Each call either solve one part of the problem or reduce the size of problem.
- The statement that solves the problem is known as **Base case**
- Every recursive algorithm must have base case.
- The rest of algorithm is known as general case.

Rules for designing Recursive Algorithm

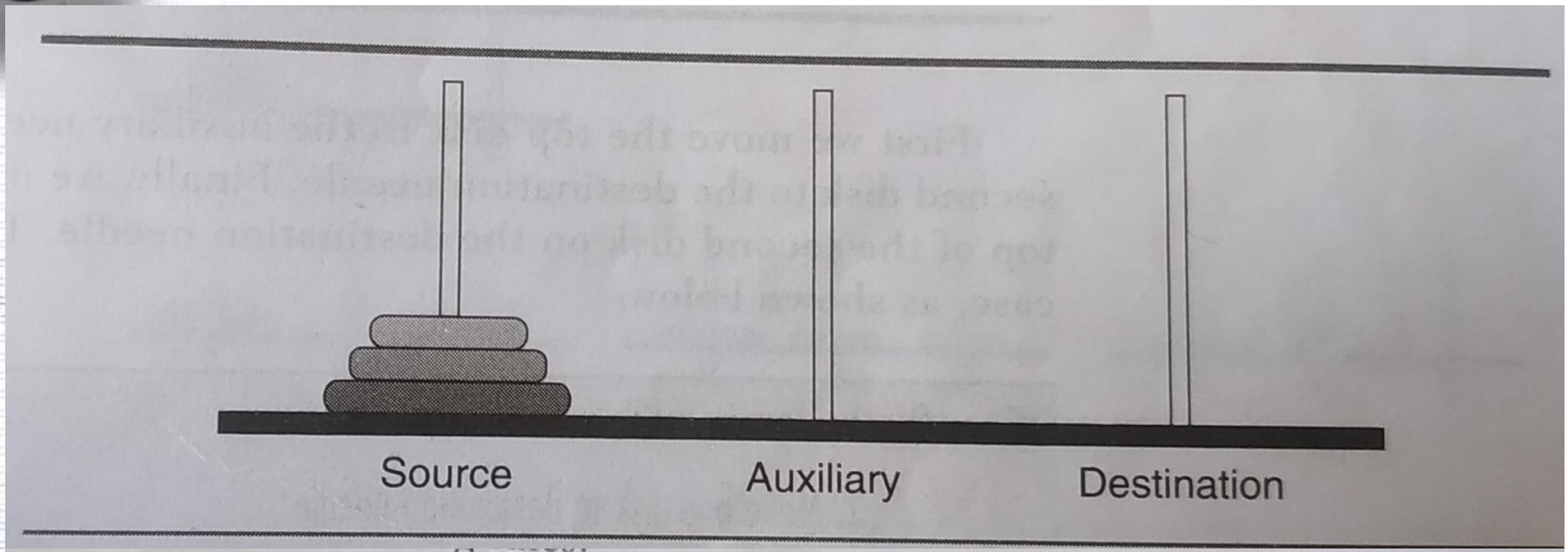
You should not use recursive algorithm if any of the following question returns ‘No’.

- Is the algorithm or data structure naturally suited for recursion?
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run within acceptable time and space limitation?

Rules for designing Recursive Algorithm

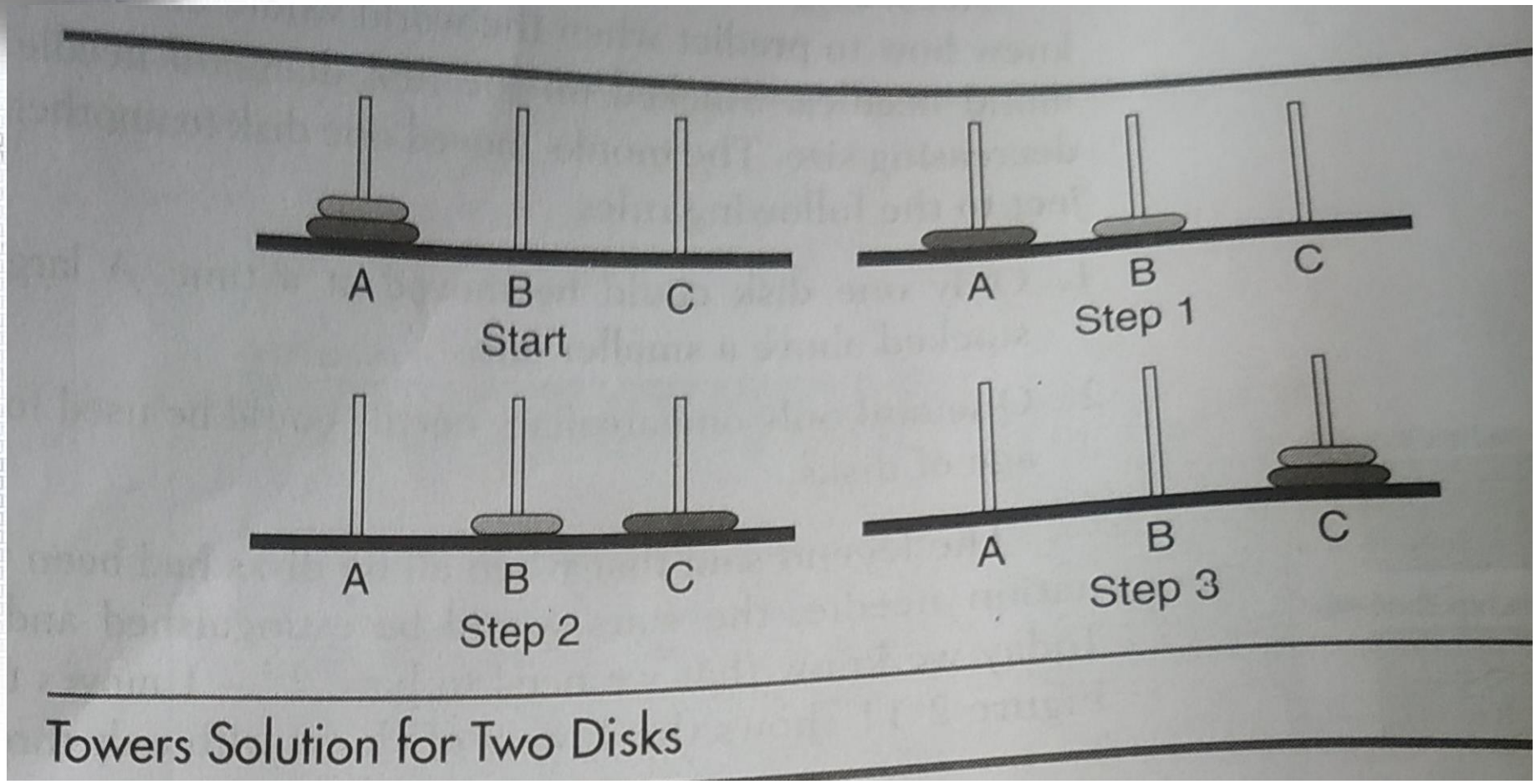
- First determine the base case
- Then determine the general case
- Combine the base case and general case into an algorithm.

Recursion – Tower of Hanoi



1. Only one disk could be moved at a time. A larger disk must never be stacked above a smaller one.
2. One and only one auxiliary needle could be used for the intermediate storage of disks.

Recursion – Tower of Hanoi

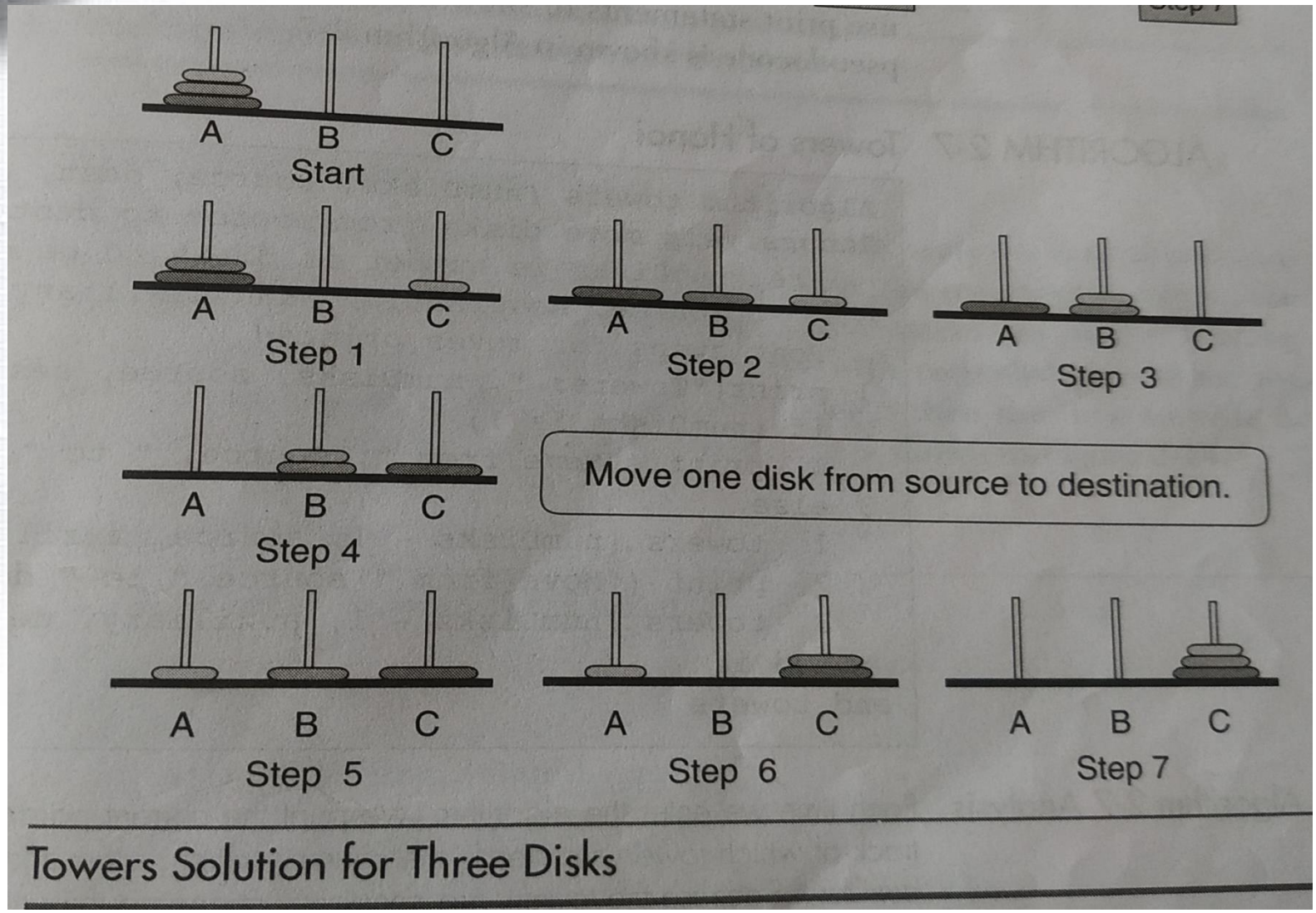


Towers Solution for Two Disks

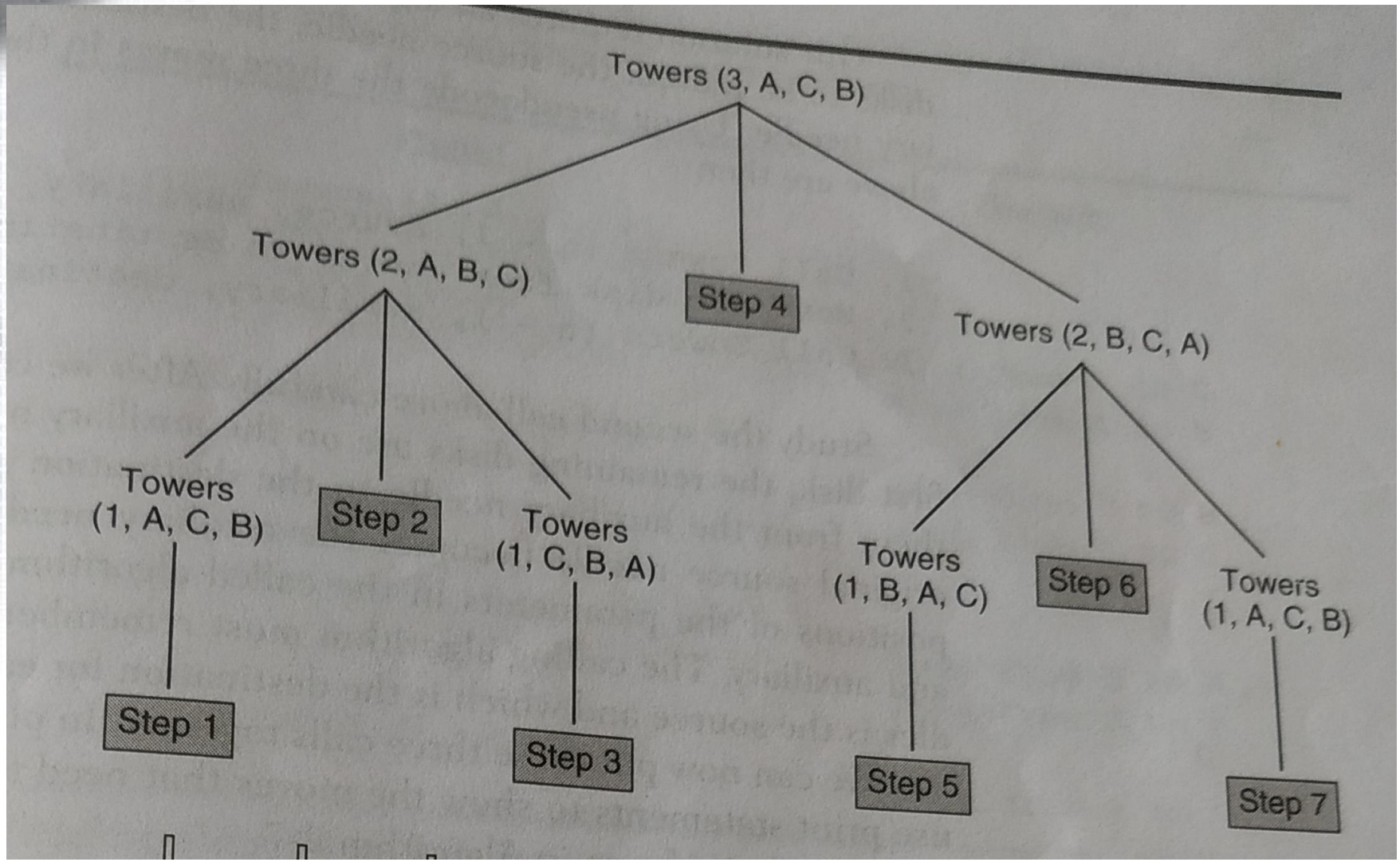
Recursion – Tower of Hanoi

1. Move one disk to auxiliary needle.
2. Move one disk to destination needle.
3. Move one disk from auxiliary to destination needle.

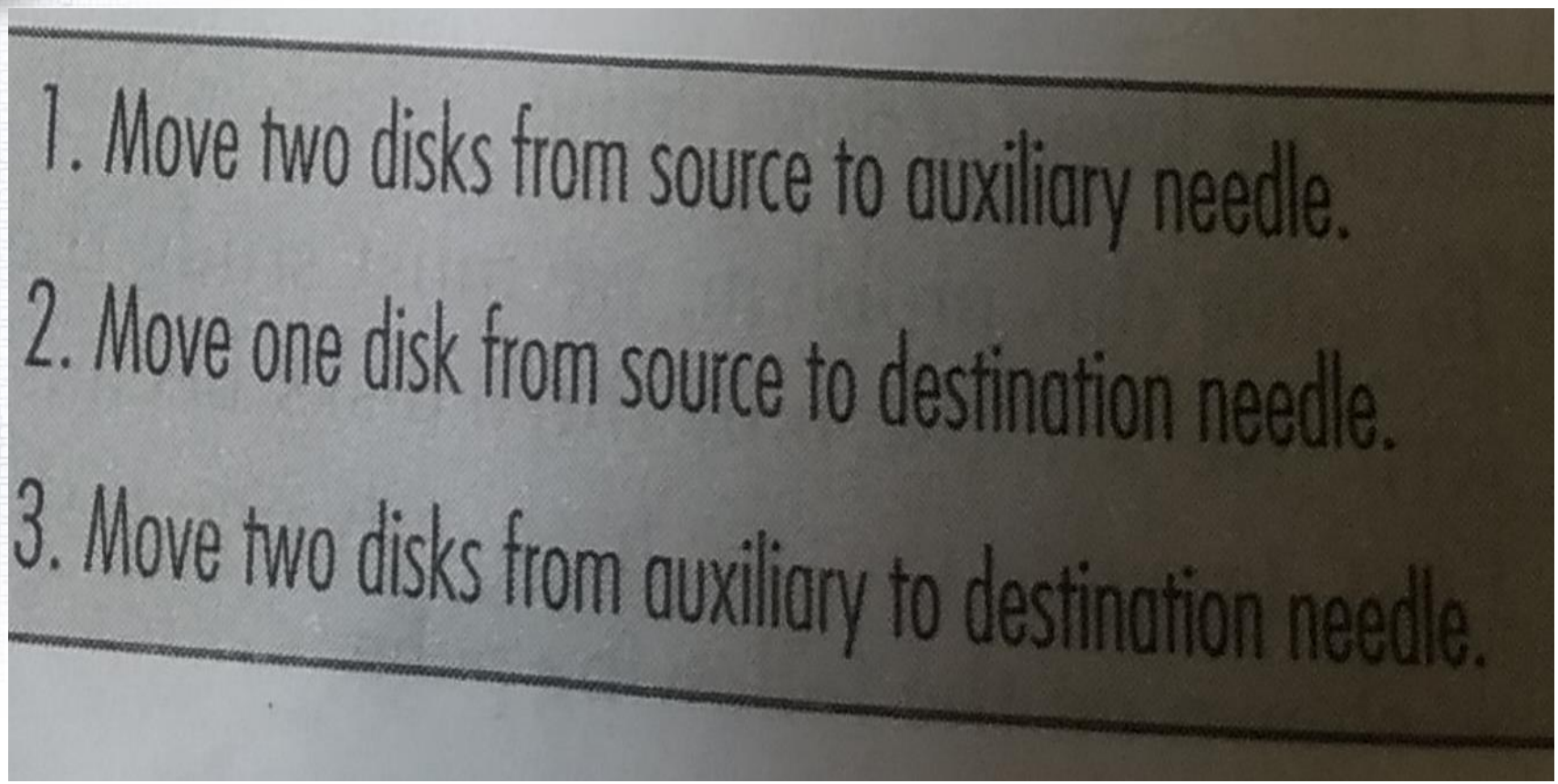
Recursion – Tower of Hanoi



Recursion – Tower of Hanoi



Recursion – Tower of Hanoi



Recursion – Tower of Hanoi Generalize Solution

1. Move *n – 1* disks from Source to auxiliary

   (General Case)
2. Move one disk from Source to destination

   (Base Case)
3. Move *n – 1* disks from auxiliary to destination

   (General Case)

Recursion – Tower of Hanoi Generalize Solution

Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
Recursively move disks from source to destination.
   Pre  numDisks is number of disks to be moved
        source, destination, and auxiliary towers given
   Post steps for moves printed
 1 print("Towers: ", numDisks, source, dest, auxiliary)
 2 if (numDisks is 1)
   1  print ("Move from ", source, " to ", dest)
 3 else
   1  towers (numDisks - 1, source, auxiliary, dest, step)
   2  print ("Move from " source " to " dest)
   3  towers (numDisks - 1, auxiliary, dest, source, step)
 4 end if
end towers
```

Recursion – Tower of Hanoi Generalize Solution

Calls:	Output:
Towers (3, A, C, B)	
Towers (2, A, B, C)	
Towers (1, A, C, B)	
	Move from A to C
	Move from A to B
Towers (1, C, B, A)	
	Move from C to B
	Move from A to C
Towers (2, B, C, A)	
Towers (1, B, A, C)	
	Move from B to A
	Move from B to C
Towers (1, A, C, B)	
	Move from A to C

Tracing Algorithm :