Design and Analysis of Algorithms (UE18CS251)

Unit I - Introduction

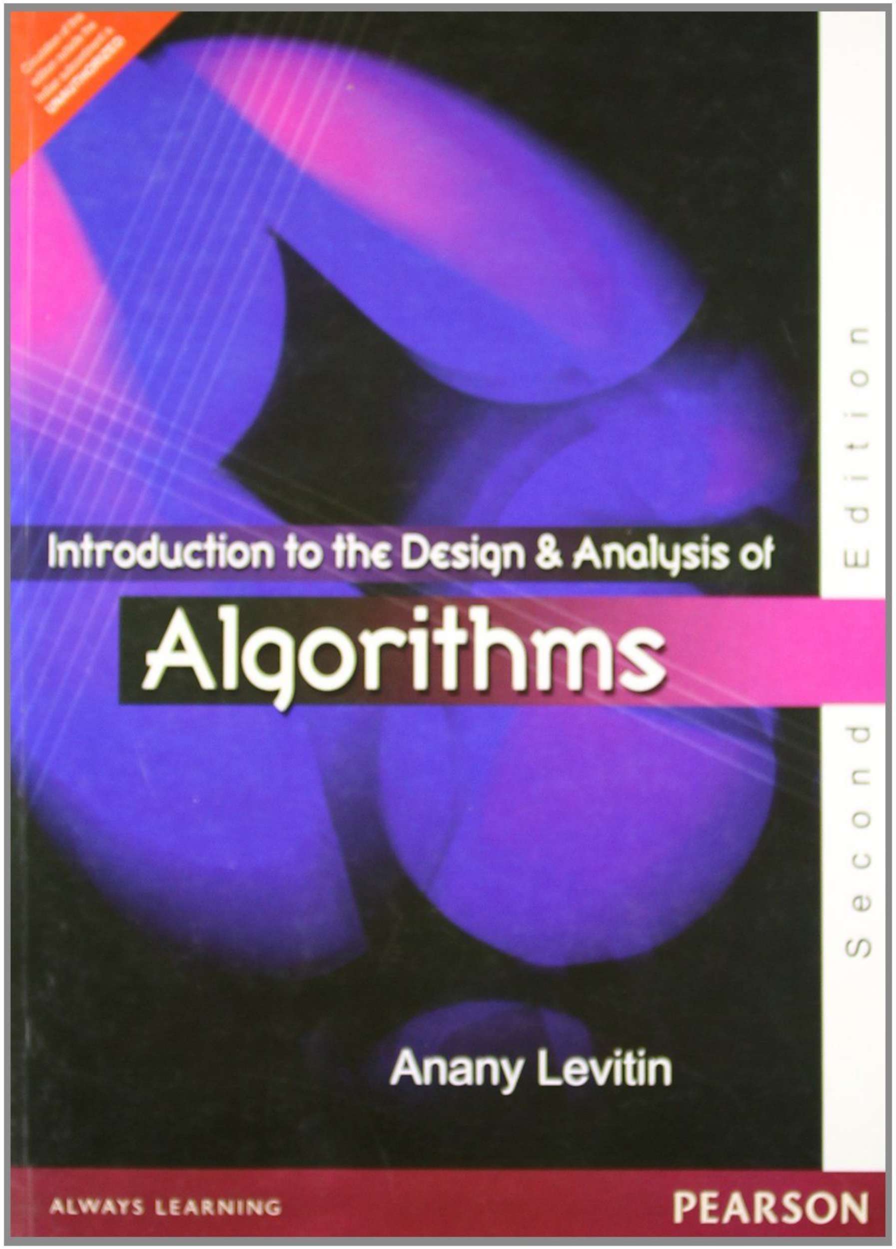
Channa Bankapur
channabankapur@pes.edu

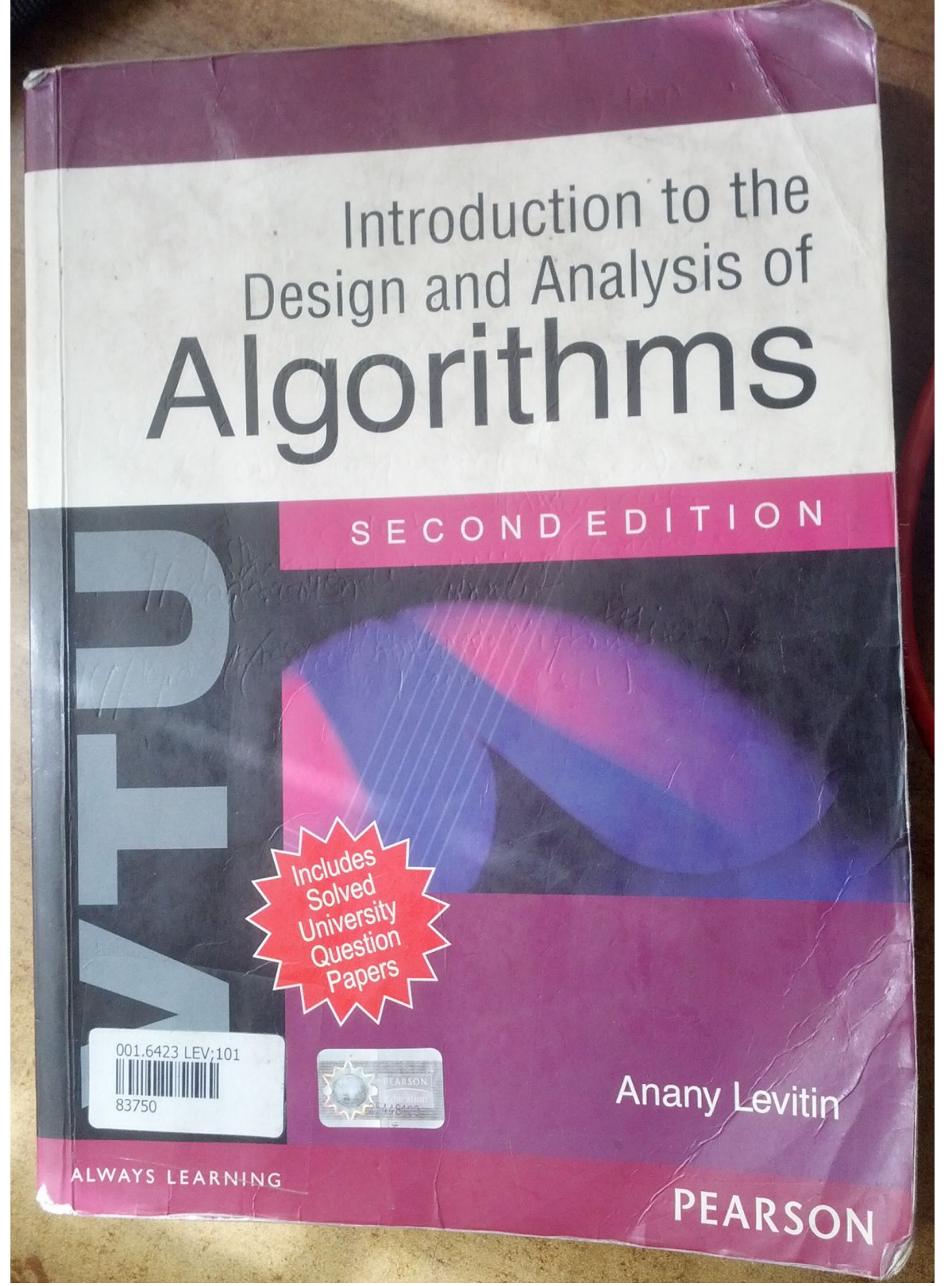
Design and Analysis of Algorithms (UE18CS251)
4 Credits, 4 lecture hours per week

ISA: 40 = 15 (T1: CBT of 40 marks) +
15 (T2: CBT of 40 marks) +
10 (Assignments).

ESA: 60 (Scaled from 100 marks theory paper)

Student Course Folder of DAA UE18CS251
tinyurl.com/UE18CS251


Introduction to the Design & Analysis of
Algorithms
Second Edition
Anany Levitin
ALWAYS LEARNING
PEARSON


Introduction to the
Design and Analysis of
Algorithms
SECOND EDITION
VTU
Includes Solved University Question Papers
001.6423 LEV;101
83750
Anany Levitin
ALWAYS LEARNING
PEARSON

Syllabus:

- **UNIT I**

  Introduction, Analysis of Algorithm Efficiency
- **UNIT II**

  Brute Force, Decrease-and-Conquer
- **UNIT III**

  Divide-and-Conquer, Transform-and-Conquer
- **UNIT IV**

  Space-and-Time Tradeoffs, Backtracking, Greedy Approach
- **UNIT V**

  Dynamic Programming, Limitations of Algorithm Power, Coping with the Limitations of Algorithm Power

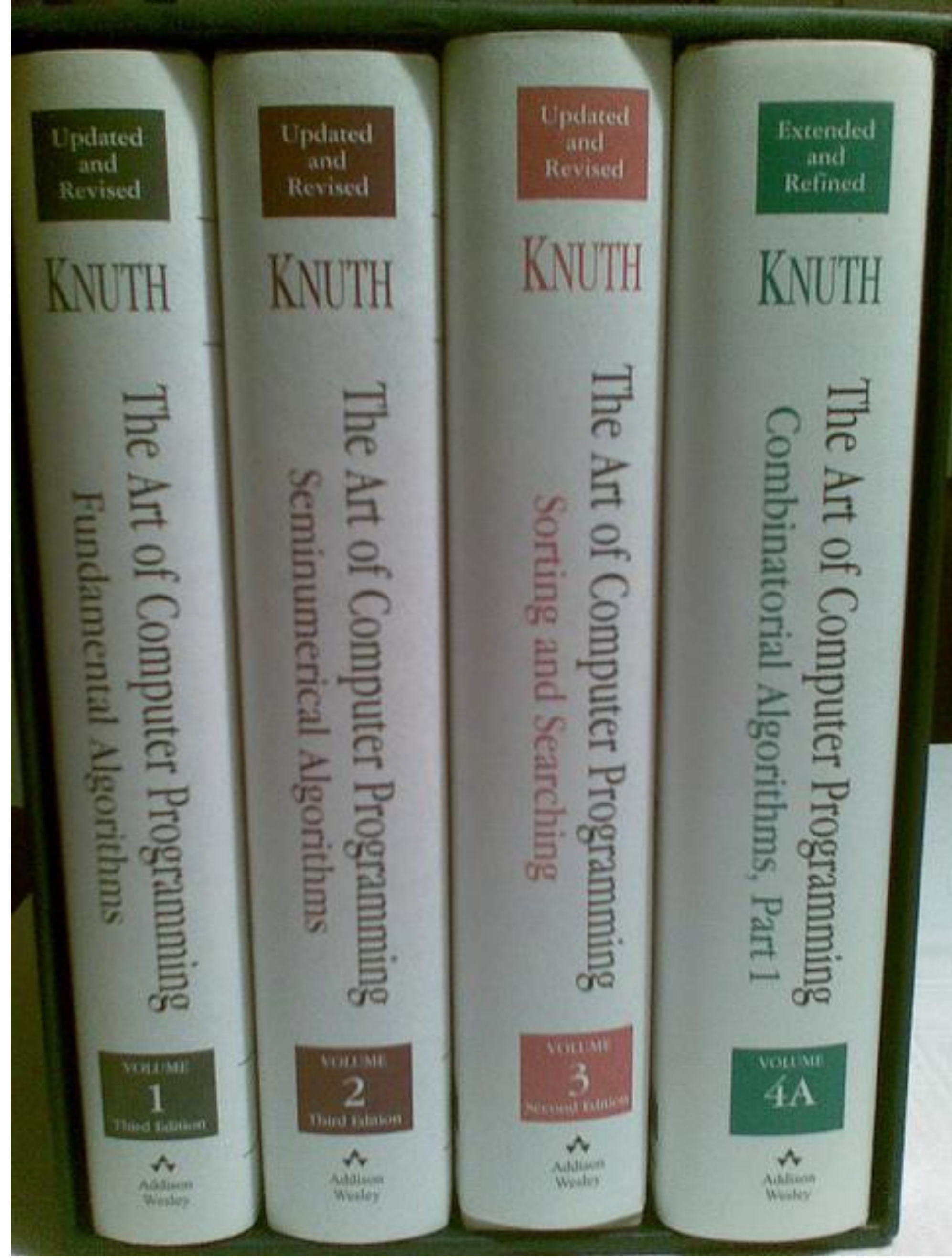

Don
Knuth

Donald Knuth
b. Jan 10, 1938

Computer Scientist
@Stanford University

Authored:
The Art of Computer Programming

Vol 1-4 Published
Vol 5 in the making
Vol 6-7 Planned



Donald Knuth said..

- A person well trained in CS knows how to deal with algorithms; how to construct them, manipulate them, understand them and analyze them. This knowledge is much more than writing good computer programs; it’s a general-purpose mental tool..

- It has often been said that a person doesn’t really understand something until after teaching it to someone else. Actually, a person doesn’t really understand something until after teaching it to a computer i.e., expressing it as an algorithm..

- An attempt to formalize things as algorithms leads to a much deeper understanding..

Why do you need to study algorithms?

1. It’s a mandatory course :-(
2. Algorithmics is core to Computer Science.
3. Computer programs wouldn’t exist without algorithms.
4. To design new algorithms and analyze their efficiency.
5. To familiarize with a standard set of important algorithms in CS.
6. To enhance your analytical skills. Algorithms can be seen as special kinds of solutions - not closed form answers, but a precisely defined procedures for getting answers.
7. Job interviews of theoretical or applied CS.

Algorithms all over the place...

- **Operating Systems:** Job Sequencing, Process Scheduling, Deadlock Avoidance, Heap Allocation, Page Replacement, Disk Scheduling…
- **Computer Networks:** Congestion Control, Huffman Coding, Encryption/Decryption, Data Encoding, Data Compression,…
- **DBMS:** B Trees, B+ Trees, Concurrency Control, Normalization, …
- **Compilers:** Parser Algorithms, Code Generation Algorithms, Symbol Table Related Algorithms, …
- **Web:** Page Ranking Algorithm, XML Parsers, …
- **Mobile:** Routing Algorithms, Address Book, …
- **Image Processing:** Edge Detection, Contrast Enhancement, Image Smoothing, …

Computer Graphics: Line Clipping, Shading, Polygon Clipping, Morphing, Animation, …

System Modeling and Simulation: Pseudo-Random Number Generators, Discrete Event Simulation Algorithms, …

Numerical Algorithms: Root Finding, ODE & PDE, Eigen Values & Eigen Vectors, Integration,…

Text Processing Algorithms: Searching, Sorting, Regular Expression Matching, Binary Search Trees,…

Operations Research: Linear Programming, Integer Programming, Scheduling, Assignment, Inventory Control,…

Game Theory: Cooperative Games, Competitive Games, Mechanism Design, …

Let’s begin..

What is an algorithm?

Definition: …

Hint: What are the common things out of a bunch of algorithms you are familiar with?

A stamp issued September 6, 1983 in the Soviet Union, commemorating **al-Khwārizmī**'s (approximate) 1200th birthday.

(CE 780 - CE 850)

Google

define algorithm

Web Images Videos News Maps More Search tools

About 10,20,00,000 results (0.50 seconds)

algorithm

/ˈalgərɪð(ə)m/

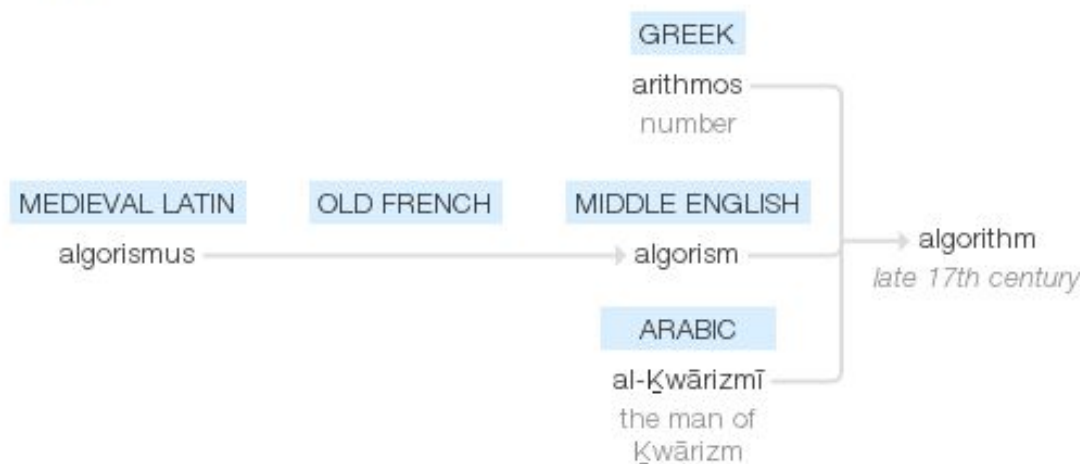
noun

noun: **algorithm**; plural noun: **algorithms**

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

"a basic **algorithm for** division"

Origin



Textbook definition of an **algorithm**:

"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."

- ➔ Instructions (a computer can understand)
- ➔ Sequence of instructions
- ➔ Unambiguous instructions
- ➔ Solving a problem
  - ◆ legitimate input
  - ◆ required output
  - ◆ finite amount of time

Algorithm for computing **GCD (m, n)**

1. …

Eg: GCD(60, 24)

12 is the GCD(60, 24).

Algorithm for computing **GCD (m, n)**

1. Assign the value of min {m, n} to t.
2. If t divides both m and n, return the value of t as the answer and stop.
3. Decrease the value of t by 1. Go to Step 2.

Eg: GCD (60, 24)

m = 60, n = 24, t = 24.

24 (doesn’t divide m), 23, 22, ..., 12.

12 divides both 60 and 24.

Hence, 12 is the GCD(60, 24).

Does it work for all the legitimate input?

Algorithm for computing **GCD (m, n)**

1. Assign the value of min {m, n} to t.
2. If the value of t is zero, return max{m, n} as the answer and stop.
3. If t divides both m and n, return the value of t as the answer and stop.
4. Decrease the value of t by 1. Go to Step 3.

Eg: GCD (55, 0)

m = 55, n = 0, t = 0.

Max{55, 0} is 55.

Hence, 55 is the GCD(55, 0).

Pseudocode of the algorithm: (Flowcharts?)

```
Algorithm GCD(m,n)
//Computes gcd(m,n) by checking consecutive integers.
//Input: Two nonnegative, not-both-zero integers m, n.
//Output: GCD of m and n.
t ← min{m, n}
if (t = 0) return max{m, n}
while (! ((m mod t = 0) and (n mod t = 0)) )
   t ← t - 1
return t
```

Euclid of Alexandria (around 300 BC)

Euclid's Algorithm uses
gcd(m,n) = gcd(n, m mod n)
and gcd(m, 0) = m

E.g.: gcd(60, 24)
= gcd(24, 12)
= gcd(12, 0) = 12

E.g.: gcd(252, 105)
= gcd(105, 42)
= gcd(42, 21)
= gcd(21, 0) = 21

Pseudocode of the Euclid's algorithm: (recursive)

```
Algorithm GCD_Euclid_Recursive(m,n)
//Computes gcd(m,n) by Euclid's algorithm.
//Input: Two nonnegative, not-both-zero integers m, n.
//Output: GCD of m and n.
if (n = 0)
   return m
return GCD_Euclid_Recursive(n, m mod n)
```

Pseudocode of the Euclid's algorithm:

```
Algorithm GCD_Euclid_Iterative(m,n)
//Computes gcd(m,n) by Euclid's algorithm.
//Input: Two nonnegative, not-both-zero integers m, n.
//Output: GCD of m and n.
while (n ≠ 0)
   r ← m mod n
   m ← n
   n ← r
endwhile
return m
```

Finding gcd(m, n) by basic principles.

E.g.: gcd(60, 24)
Prime factorization of 60: **2*****2*****3***5
Prime factorization of 24: **2*****2***2***3**
gcd(60, 24) = **2*****2*****3** = 12

E.g.: gcd(252, 105)
Prime factorization of 252: 2*2***3***3***7**
Prime factorization of 105: **3***5***7**
gcd(60, 24) = **3*****7** = 21

E.g.: gcd(3885, 1736) = ?

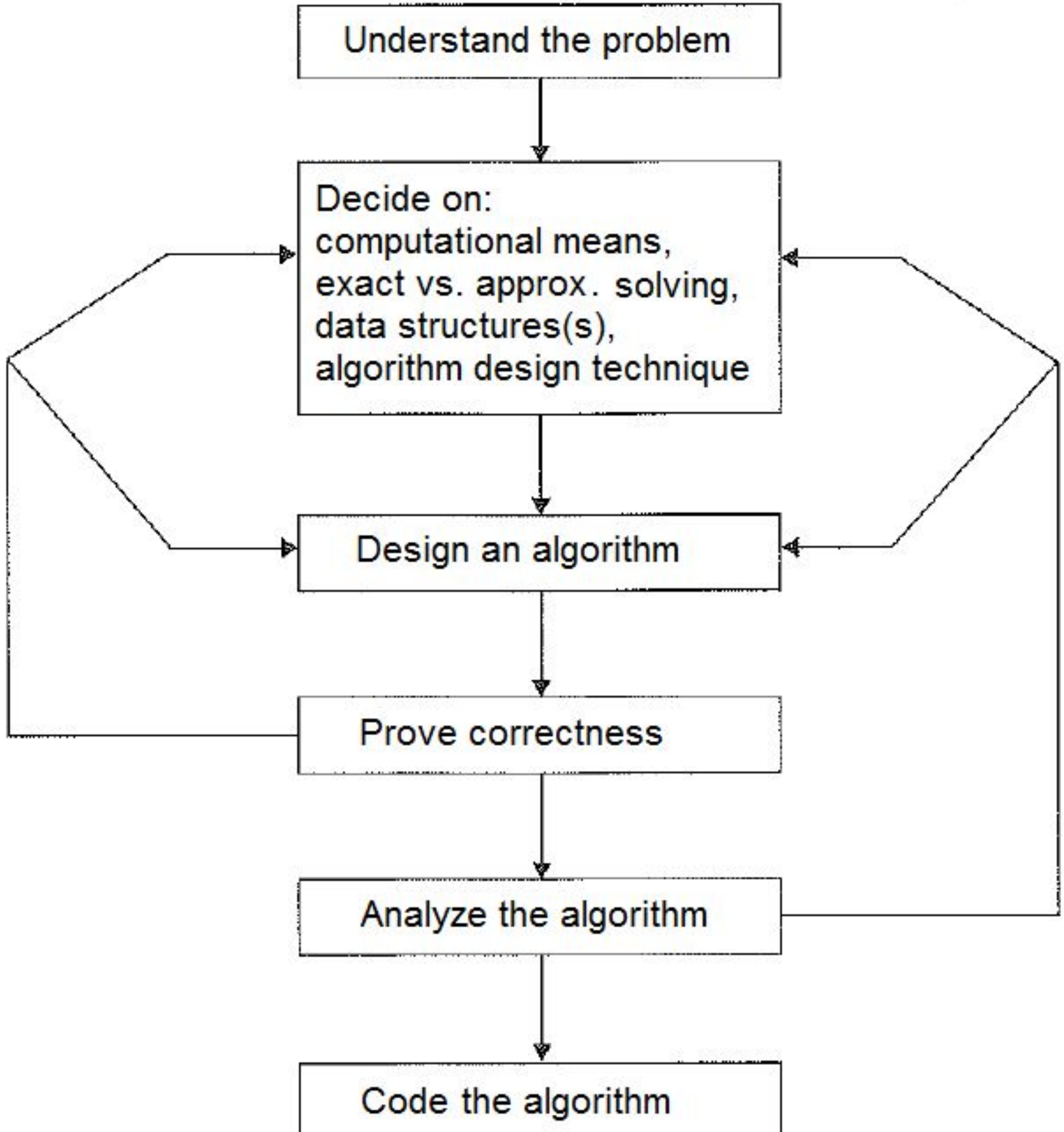
```
Algorithm GCD_byPrimeFactors(m,n)
//Input: Two nonnegative, not-both-zero integers m, n.
//Output: GCD of m and n.
if (min{m, n} = 0) return max{m, n}
if (min{m, n) = 1) return 1
M ← prime factors of m
N ← prime factors of n
T ← Common factors in M and N
if (T = Φ) p ← 1
else p ← product of the factors in T
return p
```

Does this qualify as an algorithm?

Are the steps/instructions sufficiently simple and basic?

```
Algorithm GCD_byPrimeFactors(m,n)
//Input: Two nonnegative, not-both-zero integers m, n.
//Output: GCD of m and n.
if (min{m, n} = 0) return max{m, n}
if (min{m, n) = 1) return 1
M ← prime factors of m
N ← prime factors of n
T ← Common factors in M and N
k ← number of factors in T
p ← 1
for i = 0 to k-1
   p = p * Ti
endfor
return p
```

It's an algorithm only if every instruction is implementable by the computer.


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

- Algorithms are procedural solutions to problems.
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- Boundary conditions
- Sequential vs Parallel algos.
- Exact vs Approximation algos.
- Data Structures + Algorithms = Programs
- Correctness: Not just for most the time, a correct algorithm is the one that works for all legitimate inputs.
- Time vs Space efficiency. Simplicity vs Generality.
- Coding the algorithm and tuning for the target platform.

The Course could be organized by:

- Design Techniques (Brute-Force, Dynamic Programming, Divide-and-Conquer, Greedy, etc.)
- Problem Types (Searching, Sorting, Graphs, etc.)

Important Problem Types:

1. Searching
2. String processing
3. Sorting
4. Graph problems
5. Counting problems
6. Geometric problems
7. Numerical problems

Searching:

- Finding a **search key** in a given set.
- There's a lot in between **Sequential/Linear Search** and **Binary Search**.
- And, with some kind of **preprocessing**, we can search **faster than the binary search**.
- Searching vs **Insertion/Deletion** of items.

```
Algorithm SequentialSearch(A[0..n-1], K)
//Searches for a key in an array using sequential search.
//Input: An array A[0..n-1] and a search key K.
//Output: The index of the first element of A that matches K
//    or -1 if there are no matching elements.
i ← 0
while (i < n) do
   if (A[i] = K)
      return i
   i ← i + 1
endwhile
return -1
```

```
i ← 0
while (i < n) and (A[i] ≠ K) do
   i ← i + 1
endwhile
if (i < n) return i
return -1
```

Algorithm SequentialSearch2(A[0..n-1], K)

```
//Searches for a key in an array using sequential search.
//Input: An array A[0..n-1] and a search key K.
//Output: The index of the first element of A that matches K
//    or -1 if there are no matching elements.
i ← 0
A[n] ← K
while (A[i] ≠ K) do
   i ← i + 1
endwhile
if (i < n) return i
return -1
```

```
t ← A[n-1]
A[n-1] ← K
i ← 0
while (A[i] ≠ K) do
   i ← i + 1
endwhile
A[n-1] ← t
if (i < n-1) return i
if(t = K) return n-1
return -1
```

String Processing:

- Handling non-numerical data.
- A **string** is a sequence of characters from an alphabet. **Text strings** are a kind of strings.
- **String matching** is a search problem.

String Matching:
In an **n**-character **text,** search for the first occurrence of an **m**-character **pattern**. That is, in a text of length **n,** find the first substring that matches with the pattern of length **m**.

Find **i**, the index of the leftmost character of the first matching substring in the text such that

$$
\begin{array}{ccccccccc}
t_0 & \cdots & t_i & \cdots & t_{i+j} & \cdots & t_{i+m-1} & \cdots & t_{n-1} \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & \\
 & & p_0 & \cdots & p_j & \cdots & p_{m-1} & \text{pattern } P &
\end{array}
$$

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$$

Naïve String Matching:

There are **n-m+1** substrings of length **m** in a text of length **n**. Search for the first one that matches the pattern.

```
Algorithm NaiveStringMatch(T[0..n-1],P[0..m-1])
//Implements a naive string matching.
//Input: An array T[0..n-1] of n chars representing a text
// and an array P[0..m-1] of m chars representing a pattern.
//Output: The index of the first character in the text
// that starts a matching substring
// or -1 if the search is unsuccessful.
for i ← 0 to n-m
   j ← 0
   while (j < m) and (P[j] = T[i+j]) do
      j ← j + 1
   endwhile
   if (j = m) return i
return -1
```

Sorting:

- Rearrange the list of items **in some order**.
- There must be a **total order** on the set of items.
  - Total order is a special case of **partial order** when each pair of items of the set are comparable.
- Compare on "**key**" if the item has multiple fields.
- **Stable** sorting algorithm preserves the relative order of any two equal elements in its input.
- **In-place** requires no more than constant amount of extra space.

Write an algorithm to check if the array is sorted.

```
boolean isSorted( A[0..n-1] )
//Checks if the array A is sorted.
//Input: An array A of orderable elements by ≤.
//Output: Return TRUE if array is sorted.
//        FALSE otherwise.
...
```

Write an algorithm to check if the array is sorted.

```
boolean isSorted( A[0..n-1] )
//Checks if the array A is sorted.
//Input: An array A of orderable elements by ≤.
//Output: Return TRUE if array is sorted.
//          FALSE otherwise.
for i ← 0 to n-2
   if(A[i] > A[i+1]) //not in order
      return FALSE
return TRUE
```

Sort by fixing the problems while checking for sortedness.

```
SortByCheckingSortedness( A[0..n-1] )
//Sorts by Checking sortedness.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
for i ← 0 to n-2
   if(A[i] > A[i+1])
      Swap A[i] with A[i+1]
```

Does it sort?

Sort by fixing the problems while checking for sortedness.

```
SortByCheckingSortedness2( A[0..n-1] )
//Sorts by Checking sortedness.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
while (TRUE)
   for i ← 0 to n-2
      if(A[i] > A[i+1])
         Swap A[i] with A[i+1]
   if(isSorted( A[0..n-1] ))
      return
```

Does it sort and that too in a finite amount of time?

Sort by fixing the problems while checking for sortedness.

```
SortByCheckingSortedness3( A[0..n-1] )
//Sorts by Checking sortedness.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
   for i ← 0 to n-2 //n-1 consecutive pairs
      if(A[i] > A[i+1])
         Swap A[i] with A[i+1]
   if(isSorted( A[0..n-1] ))
      return
   SortByCheckingSortedness3(A[0..n-2])
```

It should sort.

Can we move **`isSorted()`** logic into the **`for`** loop?

Algorithm SortByCheckingSortedness4(A[0..n-1])

//Sorts by an improved Bubble Sort algorithm.

//Input: An array **A** of orderable elements by ≤.

//Output: Sorted array **A**.

```
isSorted ← TRUE
for i ← 0 to n-2
   if(A[i] > A[i+1])
      Swap A[i] with A[i+1]
      isSorted ← FALSE
if(isSorted = TRUE)
   return
SortByCheckingSortedness4(A[0..n-2])
```

Base case for the recursion?

```
Algorithm SortByCheckingSortedness5(A[0..n-1])
```

//Sorts by an improved Bubble Sort algorithm.

//Input: An array **A** of orderable elements by ≤.

//Output: Sorted array **A**.

```
   if(n <= 1) return
   isSorted ← TRUE
   for i ← 0 to n-2
      if(A[i] > A[i+1])
         Swap A[i] with A[i+1]
         isSorted ← FALSE
   if(isSorted = TRUE)
      return
   SortByCheckingSortedness5(A[0..n-2])
```

Can we convert the recursive algorithm to iterative?

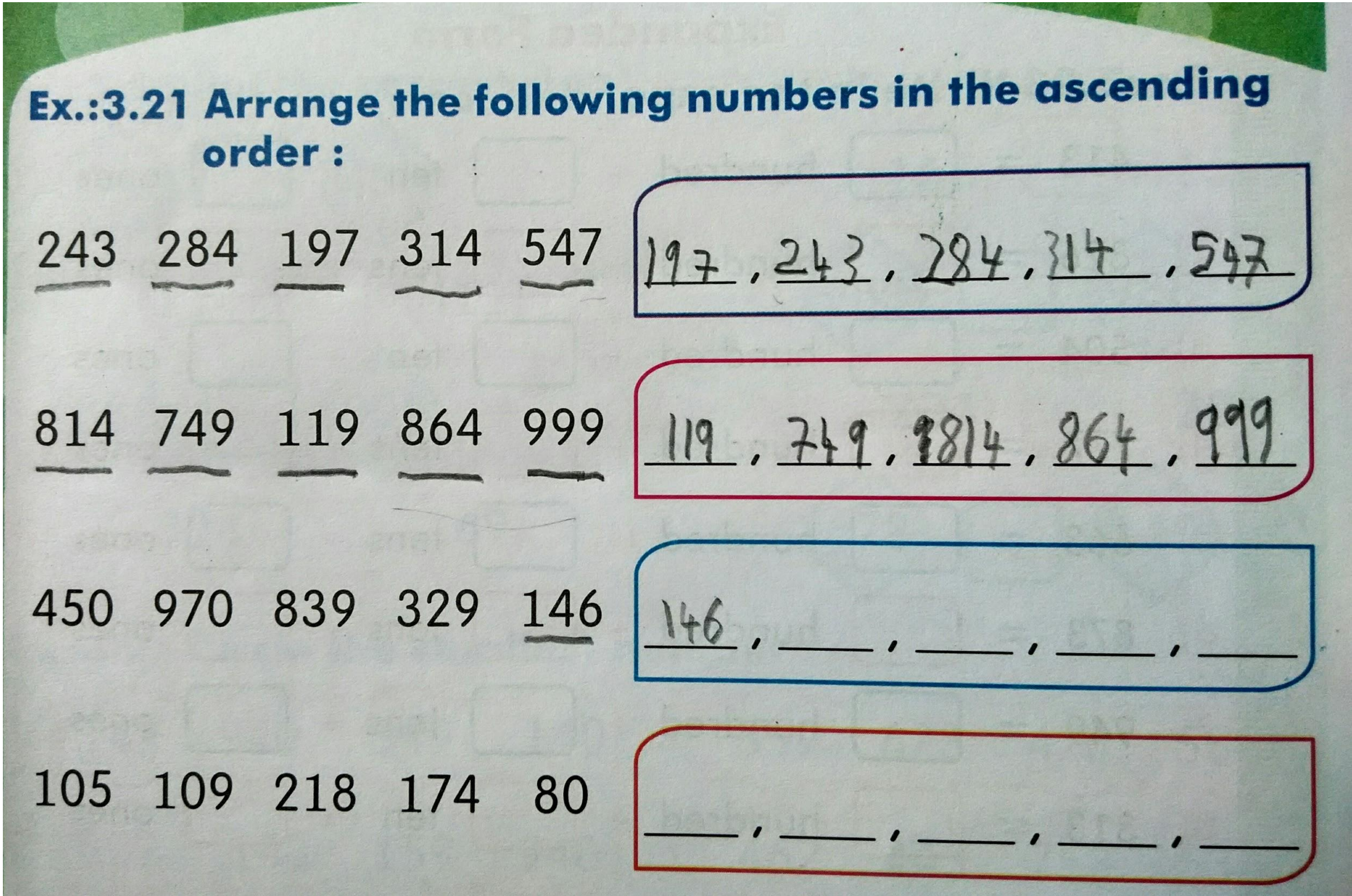
```
SortByCheckingSortedness6( A[0..n-1] )
//Sorts by Checking sortedness.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
m ← n
while(m>1)
   isSorted ← TRUE
   for i ← 0 to m-2 //m-1 consecutive pairs
      if(A[i] > A[i+1])
         Swap A[i] with A[i+1]
         isSorted ← FALSE
   if(isSorted = TRUE) return
   m--
return
```

```
Algorithm BubbleSort( A[0..n-1] )
//Sorts by an improved Bubble Sort algorithm.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
for i ← n-1 downto 1  //n-1 passes
   isSorted ← TRUE
   for j ← 0 to i-1  //last n-i-1 elements are sorted
      if(A[j] > A[j+1])
         Swap A[j] with A[j+1]
         isSorted ← FALSE
   if(isSorted = TRUE)
      return
return
```

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

```
Algorithm BubbleSort( A[0..n-1] )
//Sorts by Bubble Sort algorithm.
//Input: An array A of orderable elements by ≤.
//Output: Sorted array A.
for i ← n-1 downto 1  //n-1 passes
   for j ← 0 to i-1  //last n-i-1 elements are sorted
      if(A[j] > A[j+1])
         Swap A[j] with A[j+1]
return
```

How does a kindergarten kid **sort** ?



Ex.:3.21 Arrange the following numbers in the ascending order :

243 284 197 314 547 — 197 , 243 , 284 , 314 , 547

814 749 119 864 999 — 119 , 749 , ~~9~~814 , 864 , 999

450 970 839 329 146 — 146 , ____ , ____ , ____ , ____

105 109 218 174 80 — ____ , ____ , ____ , ____ , ____

Selection Sort:
Find the smallest of the unsorted array and place it at the beginning of the unsorted array. Reduce the unsorted array by excluding the first one, which is already in its final position. Repeat sorting the unsorted array as long as there is only one element left in the unsorted array.

```
Algorithm SelectionSort_Recursive(A[0..n-1])
//Sorts a given array by Selection Sort.
//Input: An array A[0..n-1] of orderable elements.
//Output: Array A[0..n-1] sorted in ascending order.
   if(n ≤ 1) return
   min ← minIndex(A[0..n-1])
   Swap A[0] with A[min]
   SelectionSort_Recursive(A[1..n-1])
```

Selection Sort:
Find the smallest of the unsorted array and place it at the beginning of the unsorted array. Reduce the unsorted array by excluding the first one, which is already in its final position.

```
Algorithm SelectionSort(A[0..n-1])
//Sorts a given array by Selection Sort.
//Input: An array A[0..n-1] of orderable elements.
//Output: Array A[0..n-1] sorted in ascending order.
for i ← 0 to n-2
   min ← minIndex(A[i..n-1])
   Swap A[i] with A[min]
```

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions — the last $n - i$ elements

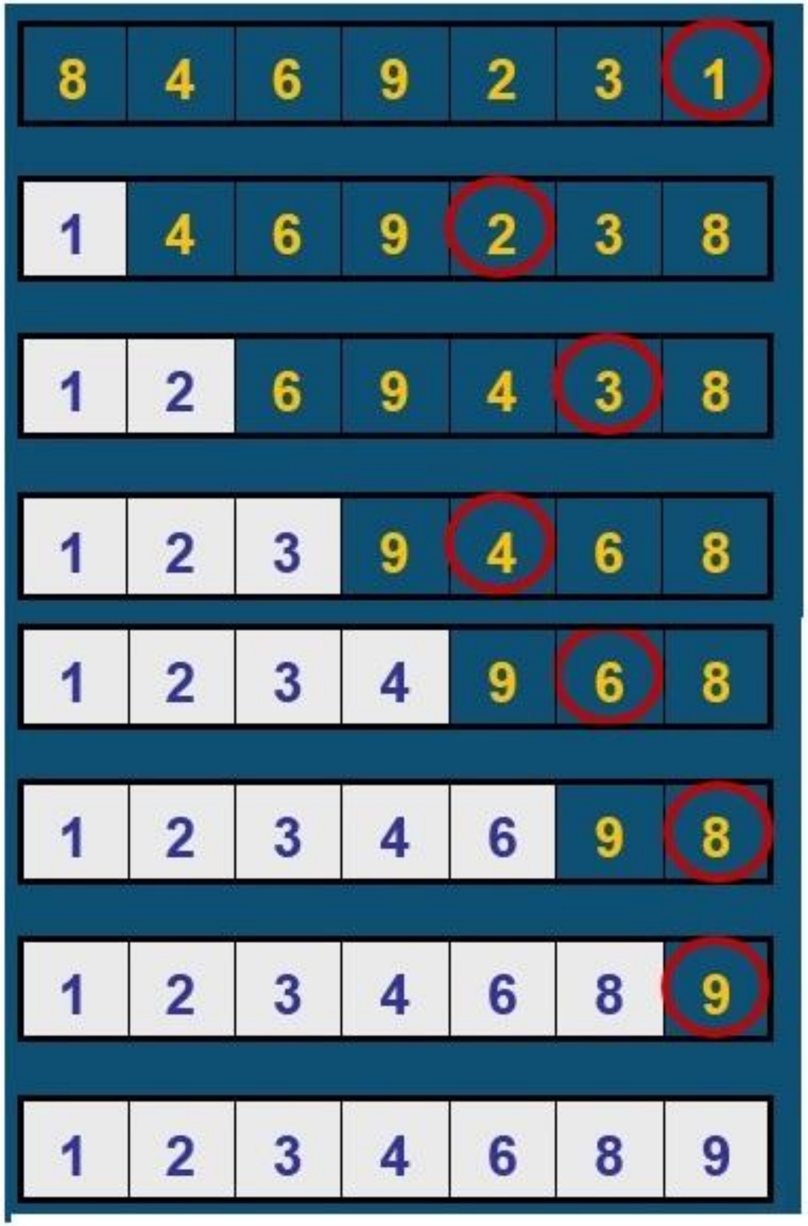
```
Algorithm SelectionSort(A[0..n-1])
//Sorts a given array by Selection Sort.
//Input: An array A[0..n-1] of orderable elements.
//Output: Array A[0..n-1] sorted in ascending order.
for i ← 0 to n-2
   min ← i
   for j ← i+1 to n-1
      if(A[j] < A[min]) min ← j
   Swap A[i] with A[min]
return A
```

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions — the last $n - i$ elements

Selection Sort:

Example: 8 4 6 9 2 3 1



Graph Problems:

Many problems in computer science can be **modelled as a graph** and solved using well-known **graph processing algorithms**.

- Graph traversal
- Shortest path
- Topological sorting
- Spanning trees
- Travelling salesperson problem (TSP)
- Graph-coloring
- Web graph

Travelling Salesman Problem:

1. Bengaluru
2. New Delhi
3. Mumbai
4. Chennai
5. Kolkata
6. Kochi
7. Hyderabad
8. Bhopal
9. Udaipur
10. Raipur

Travelling Salesman Problem:

1. Given *n* cities and distances between each pair of cities, find the **shortest round trip** that visits all other cities (and returns to the origin city).
2. It's essentially finding the shortest ***Hamiltonian circuit***.

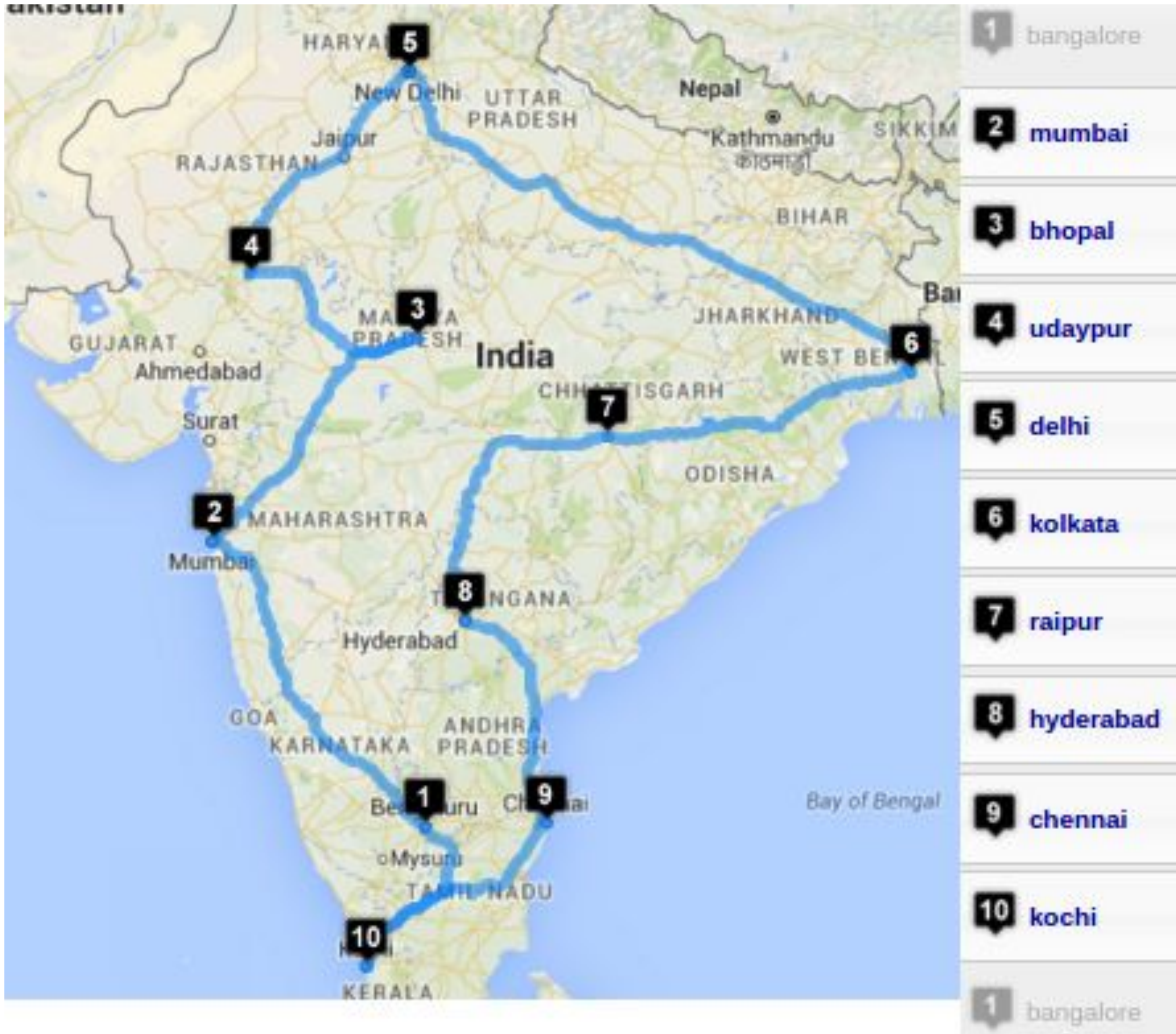
Eg: Driving time between some 10 cities of India (Cost Matrix).

000000	110189	050573	020948	109480	034435	028433	074836	091767	068406
109006	000000	079663	118195	079397	143304	083593	045792	037923	068146
051516	080265	000000	070149	121881	083636	044745	043763	042416	067450
021557	119539	069838	000000	095820	042397	037471	084186	111032	077756
110053	081231	121373	095977	000000	134475	085826	087690	100264	054016
034488	144238	082769	041728	134042	000000	062482	108885	123963	102455
028473	084770	045153	037117	085732	062772	000000	049417	078006	042987
075056	046162	044536	084245	086579	109354	049641	000000	031151	038399
092933	037994	042414	111566	099497	125053	078960	031010	000000	068113
068718	068844	068336	077907	055357	103016	043305	038648	068634	000000

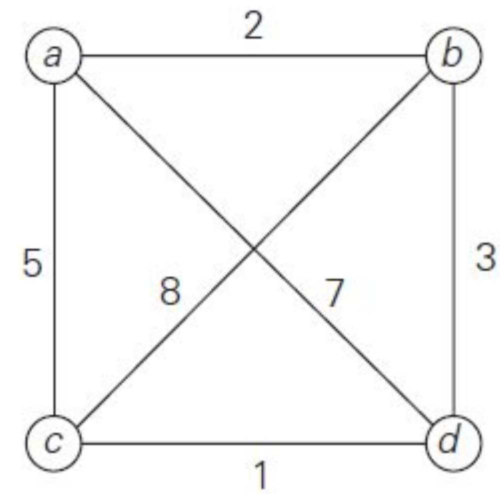
Travelling Salesman Problem:

1. Bengaluru
2. New Delhi
3. Mumbai
4. Chennai
5. Kolkata
6. Kochi
7. Hyderabad
8. Bhopal
9. Udaipur
10. Raipur

Shortest round trip takes **454201** sec.



Travelling Salesman Problem:

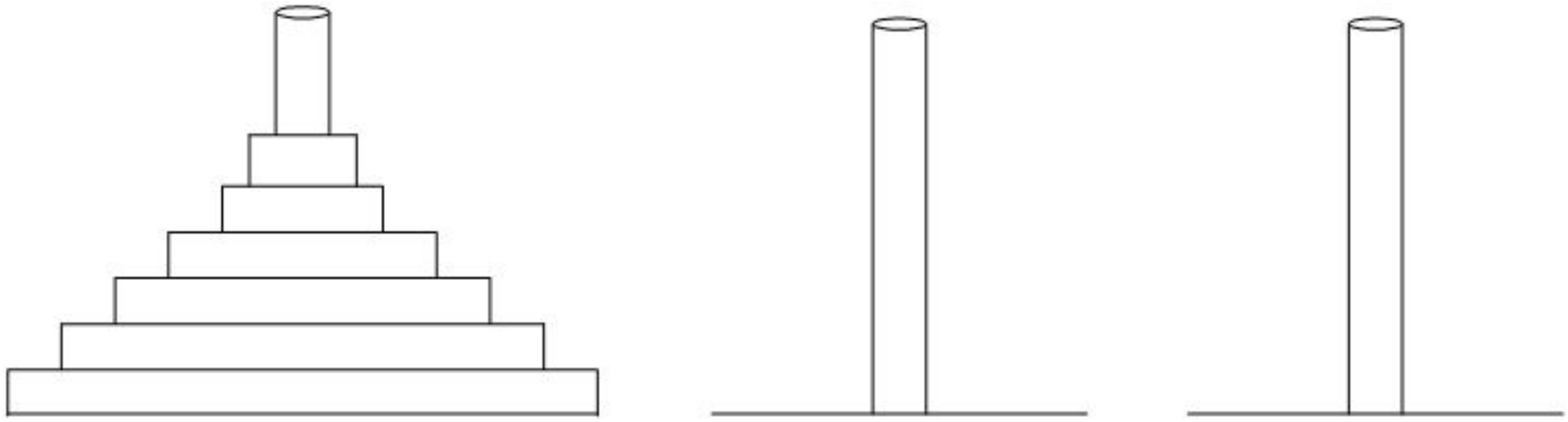


Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

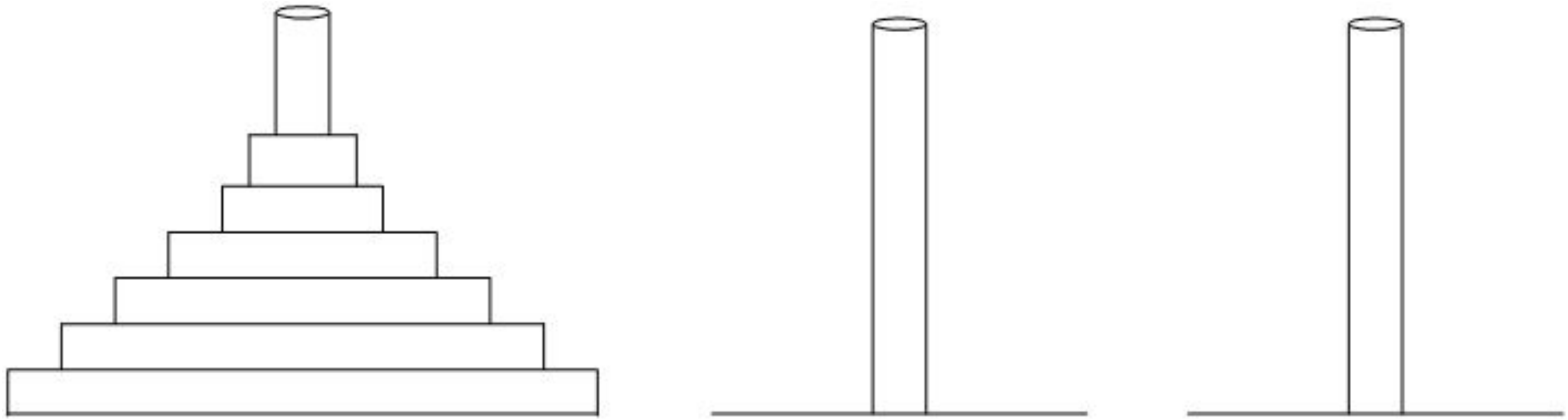
```
ALGORITHM Travelling Salesman Problem
//Input: nxn adjacency matrix A.
//Output: Cost of min-cost Hamiltonian circuit.
mincost ← INFINITY
for each permutation of n-1 cities
   cost ← 0
   for each edge in the Hamiltonian circuit
      //formed by the permutation
      cost ← cost + (cost of the edge)
   endfor
   if (cost < mincost) mincost ← cost
endfor
return mincost
```

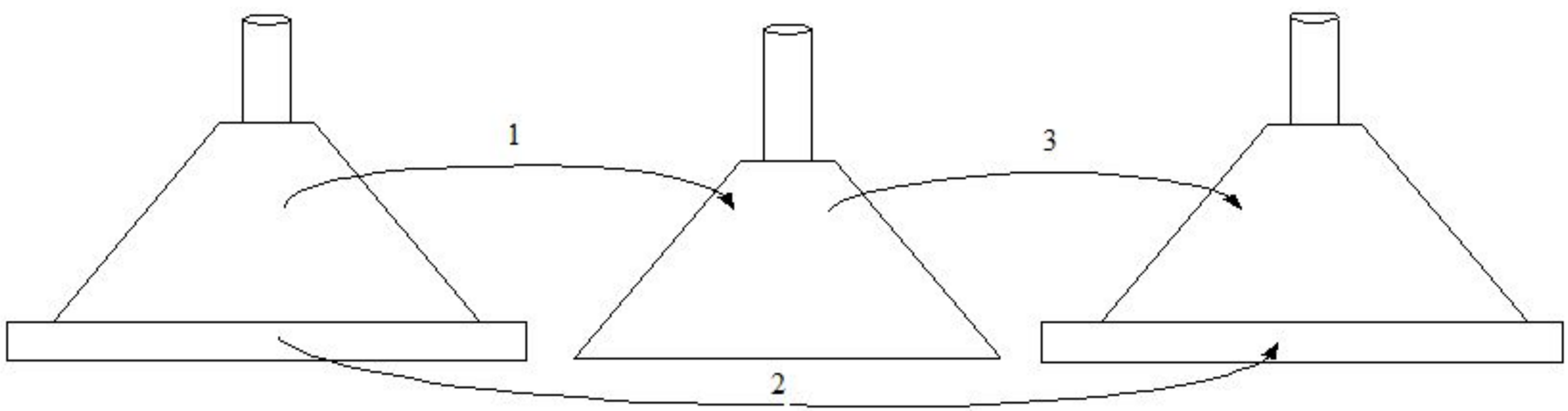
```
ALGORITHM Travelling Salesman Problem
//Input: n x n adjacency matrix A. Assumed n > 1.
//Output: Cost of min-cost Hamiltonian circuit.
//getNextPermutation(P[]) returns true with next permn
//in lexicographic order if it exists, false otherwise.
mincost ← INFINITY
Perm[0..n-2] ← [1, 2, 3, …, n-1] //1st permn.
do
  cost ← A[0, Perm[0]] //1st edge of the ckt
  cost ← cost + A[Perm[n-2], 0] //last edge
  for i ← 0 to n-3
    cost ← cost + A[Perm[i], Perm[i+1]]
  if (cost < mincost) mincost ← cost
while(getNextPermutation(Perm[0..n-2]))
return mincost
```

Tower of Hanoi puzzle:



Tower of Hanoi puzzle:





```
Algorithm Hanoi(n, Src, Dest, Int)
//Move n disks from Src peg to Dst peg as per the Tower
of Hanoi puzzle.
//Input: n (nonnegative int) disks and three pegs.
//Output: Movement of disks between pegs in the order
// of solving the puzzle.
   if (n = 0) return
   Hanoi(n-1, Src, Int, Dst)
   Move disk# n from Src to Dst
   Hanoi(n-1, Int, Dst, Src)
   return
```

</ Introduction to Algorithms >