DAA

- Algorithm

- Algorithm

- Finite set of steps involved to solve a problem is called algorithm.

- Characteristics of fundamental instructions:
- Definiteness
- Finiteness
- Deterministic
- Input/Output

Types of Problems

- Trackable
- Intrackable
- Decision
- Optimization

- Trackable : Problems that can be solvable in a reasonable(polynomial) time.
- Intrackable : Some problems are intractable, as they grow large, we are unable to solve them in reasonable time.

Tractability

- What constitutes reasonable time?
– Standard working definition: polynomial time
– On an input of size n the worst-case running time is O(n k) for some constant k
– $O(n^2)$, $O(n^3)$, O(1), O(n lg n), $O(2^n)$, $O(n^n)$, O(n!)
– Polynomial time: $O(n^2)$, $O(n^3)$, O(1), O(n lg n)
– Not in polynomial time: $O(2^n)$, $O(n^n)$, O(n!)

- Are all problems solvable in polynomial time?
- No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given.

Optimization or Decision Problems

- Optimization Problems

– An optimization problem is one which asks, “What is the optimal solution to problem X?”

– Examples:

- 0-1 Knapsack
- Fractional Knapsack
- Minimum Spanning Tree

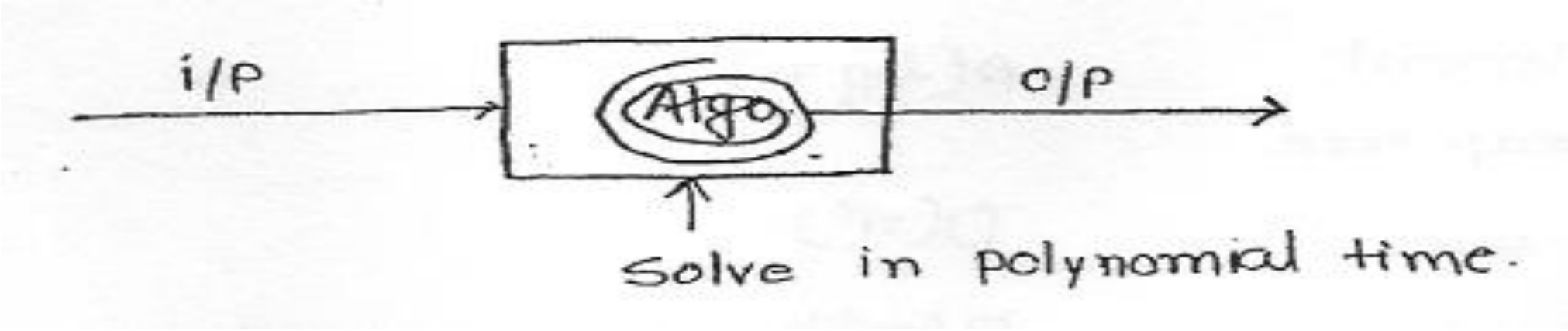
- Decision Problems
  - A decision problem is one with yes/no answer
  - Examples:
    - Does a graph G have a MST of weight <= W?

The Class P

- P: the class of problems that have polynomial-time deterministic algorithms.

– That is, they are solvable in O(p(n)), where p(n) is a polynomial on n

– A deterministic algorithm is (essentially) one that always computes the correct answer



Examples of P class problems

- Fractional Knapsack
- MST
- Sorting

The class NP

- NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)

or

- Problem that can be verified in polynomial time using non-deterministic turing machine.

– (A determinstic computer is what we know)

– A nondeterministic computer is one that can “guess” the right answer or solution

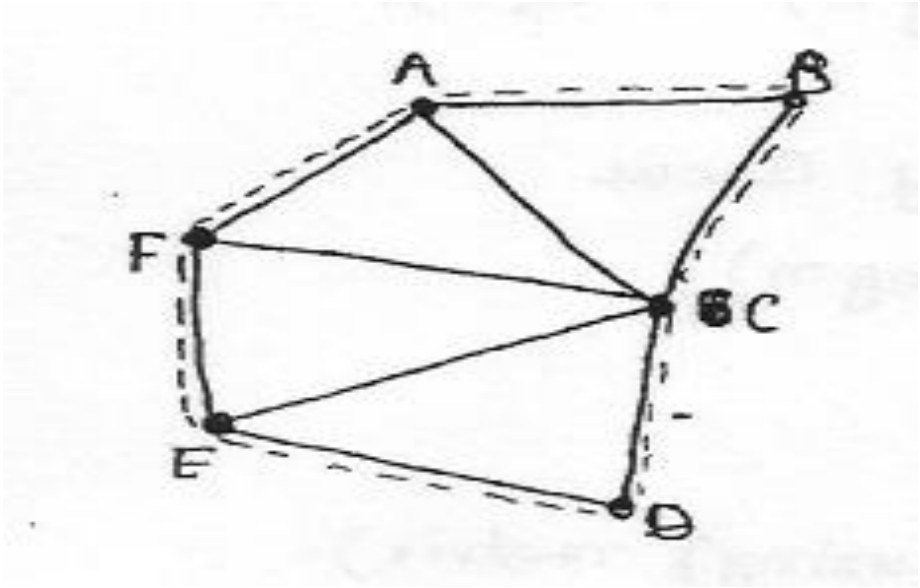
*Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes

*Thus NP can also be thought of as the class of problems “whose solutions can be verified in polynomial time”

Example Problems in NP

– Traveling Salesman
– Graph Coloring
– Satisfiability (SAT)

- Example of an NP problem: The Hamiltonian Cycle (HC) problem
  - Input: A graph G
  - Question: Goes G have a Hamiltonian Cycle?



Hamiltonian Path: AB – BC – CD – DE – EF

- How do we prove whether the problem belongs to NP class or not?

Prover	Verifier
The graph contains Hamiltonian path of length 5.	How do you say?
Pass through all the edges which forms Hamiltonian path AB – BC – CD – DE – EF	Now verifier verifies it in O(n-1) = O(n) And says 'Yes'

- Since Hamiltonian path problem verified in polynomial time. So, it is called as an NP-class problem.
- Note: Problems that can be solved in polynomial time also verified in polynomial time. $P \subseteq NP$.

NP-complete problems

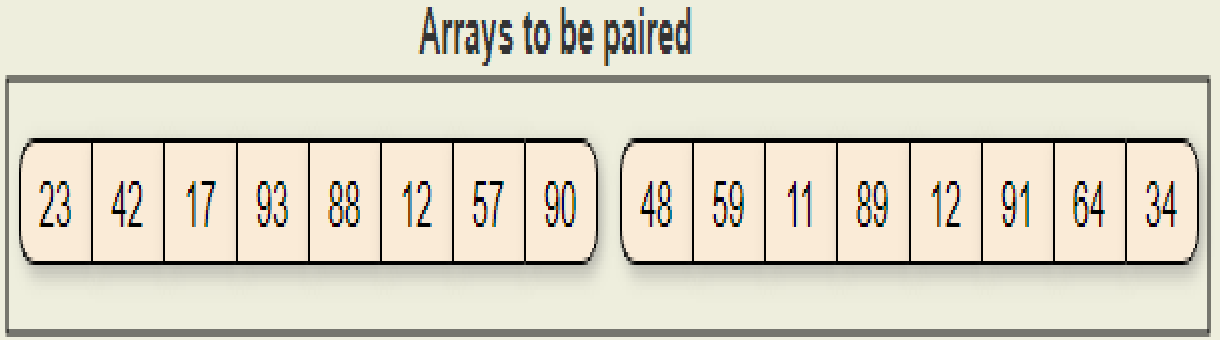
- A problem is NP-complete if the problem is both
  - NP-hard, and
  - NP

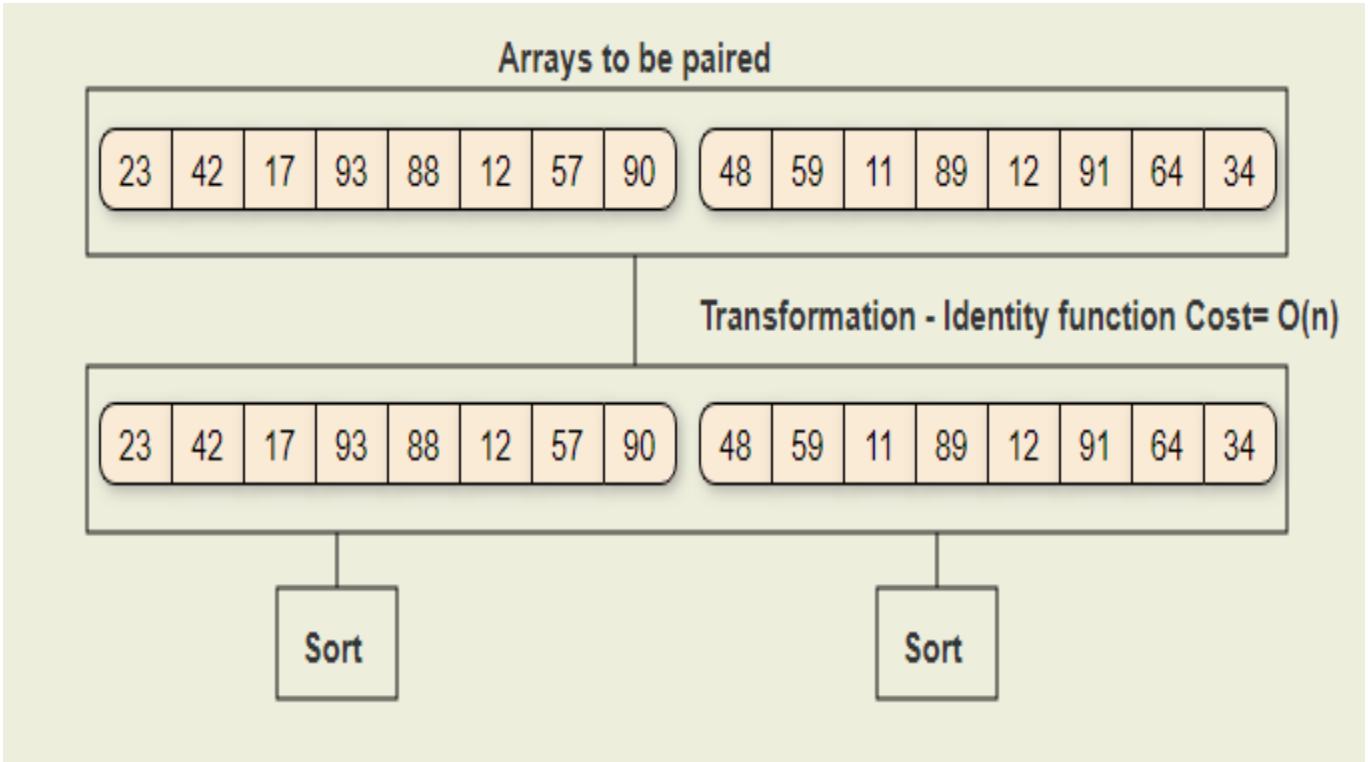
Reduction

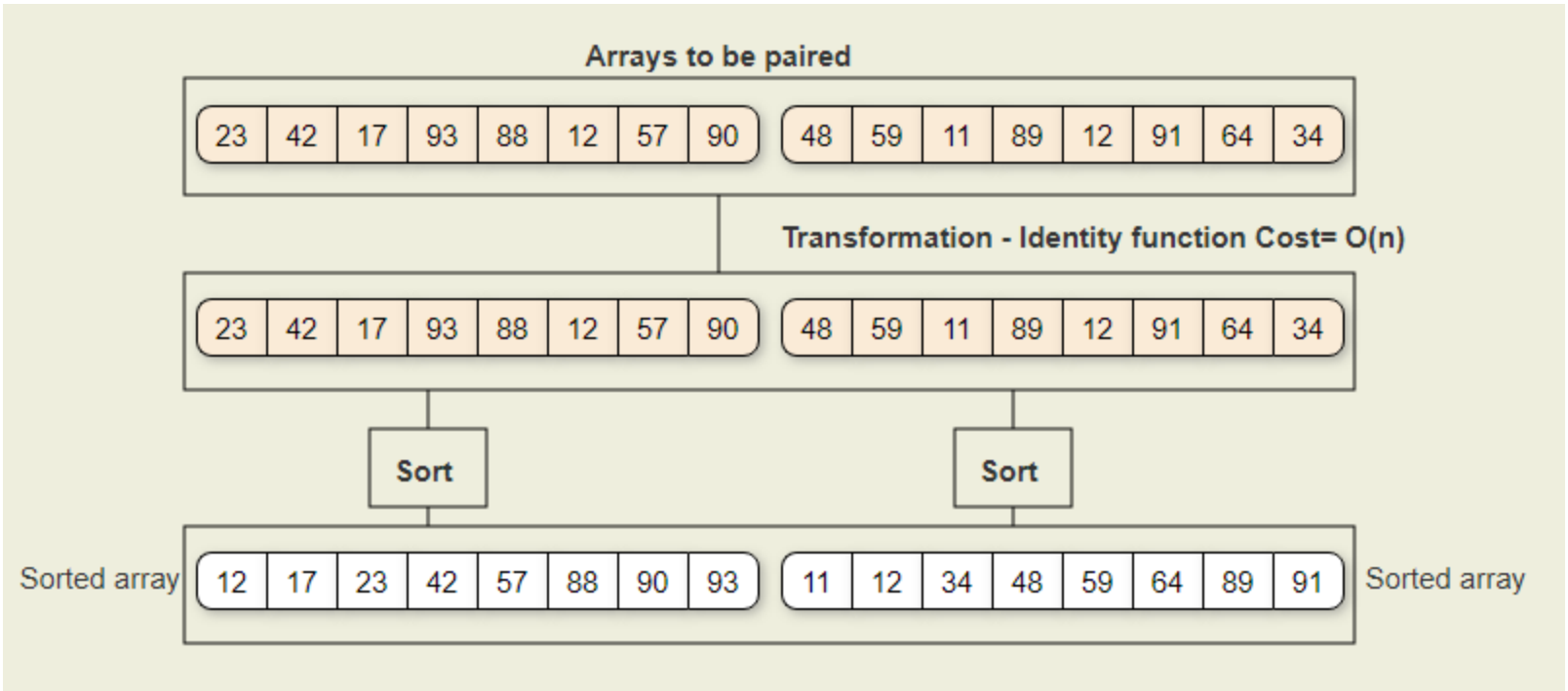
- A problem R can be reduced to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R

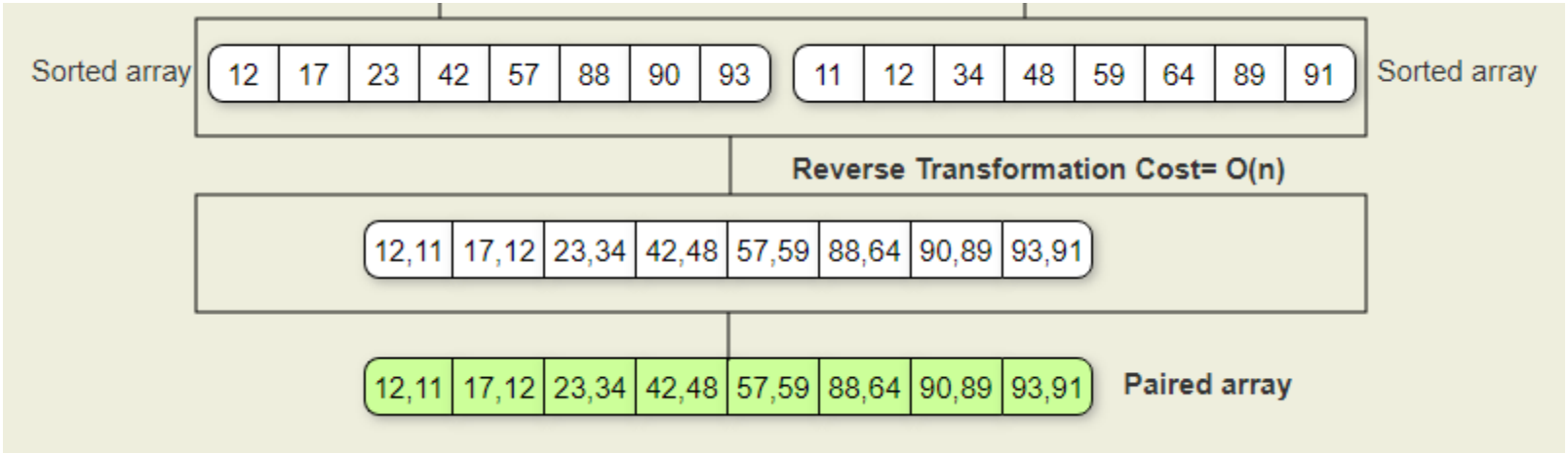
- PAIRING
- Input: Two sequences of integers $X=(x0,x1,...,xn-1)$ and $Y=(y0,y1,...,yn-1)$.
- Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, the next least value in X is paired with the next least value in Y, and so on.

Pairing of two arrays by reduction to sorting



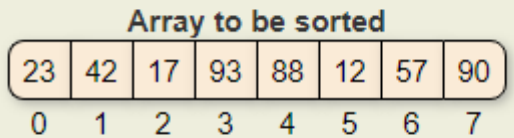

Arrays to be paired
23 42 17 93 88 12 57 90
48 59 11 89 12 91 64 34
Transformation - Identity function Cost= O(n)
23 42 17 93 88 12 57 90
48 59 11 89 12 91 64 34
Sort
Sort

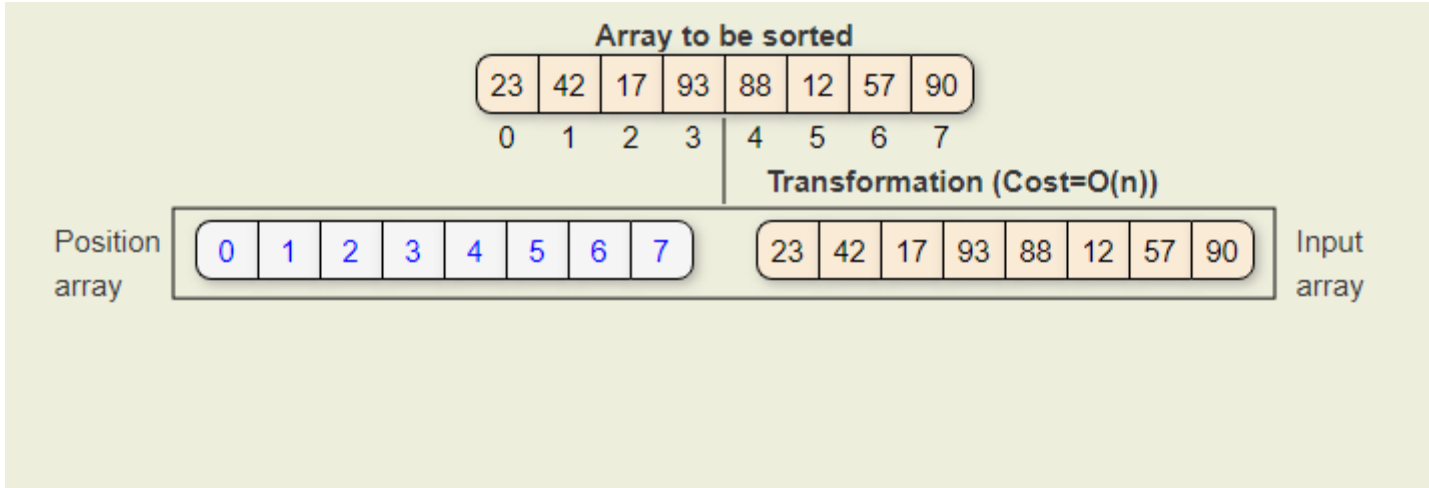

Arrays to be paired
23 42 17 93 88 12 57 90
48 59 11 89 12 91 64 34
Transformation - Identity function Cost= O(n)
23 42 17 93 88 12 57 90
48 59 11 89 12 91 64 34
Sort
Sort
Sorted array
12 17 23 42 57 88 90 93
11 12 34 48 59 64 89 91
Sorted array

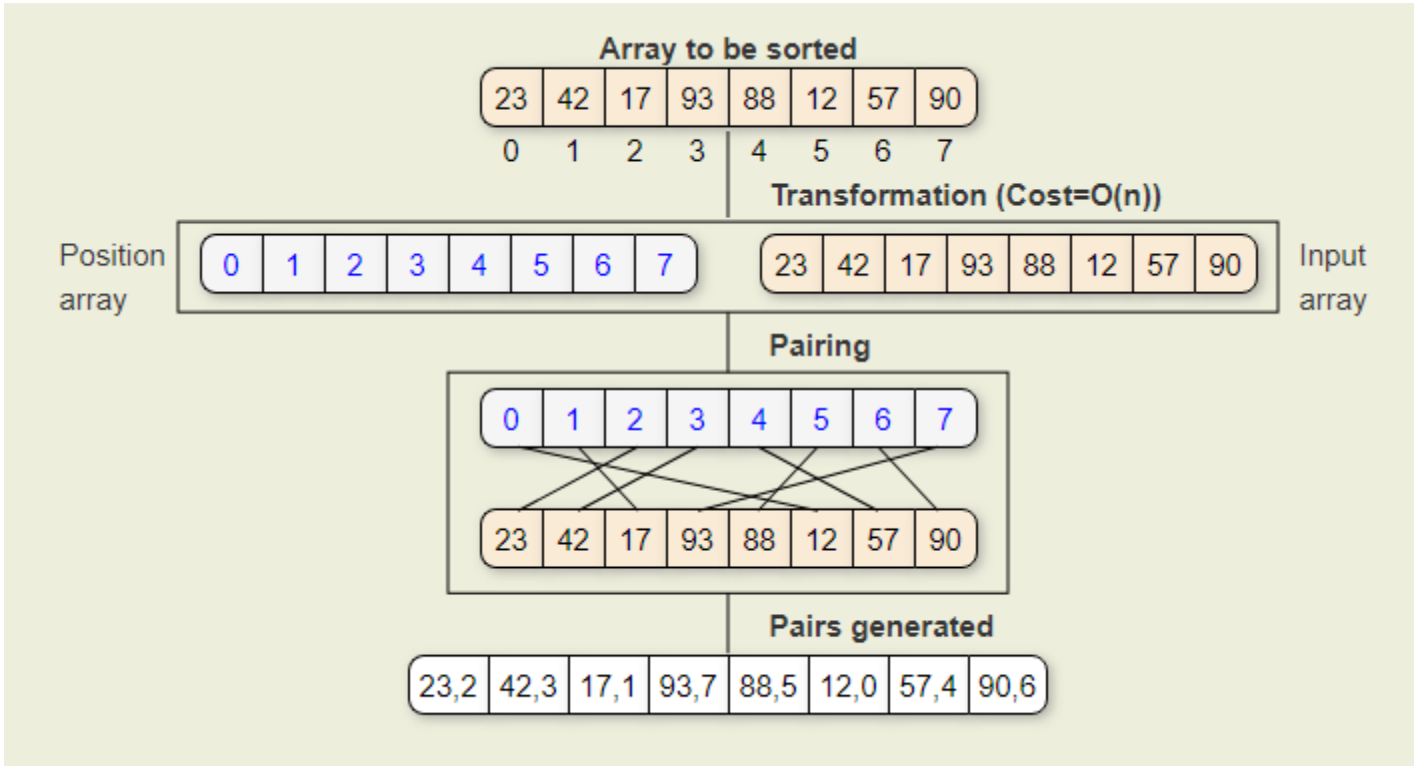


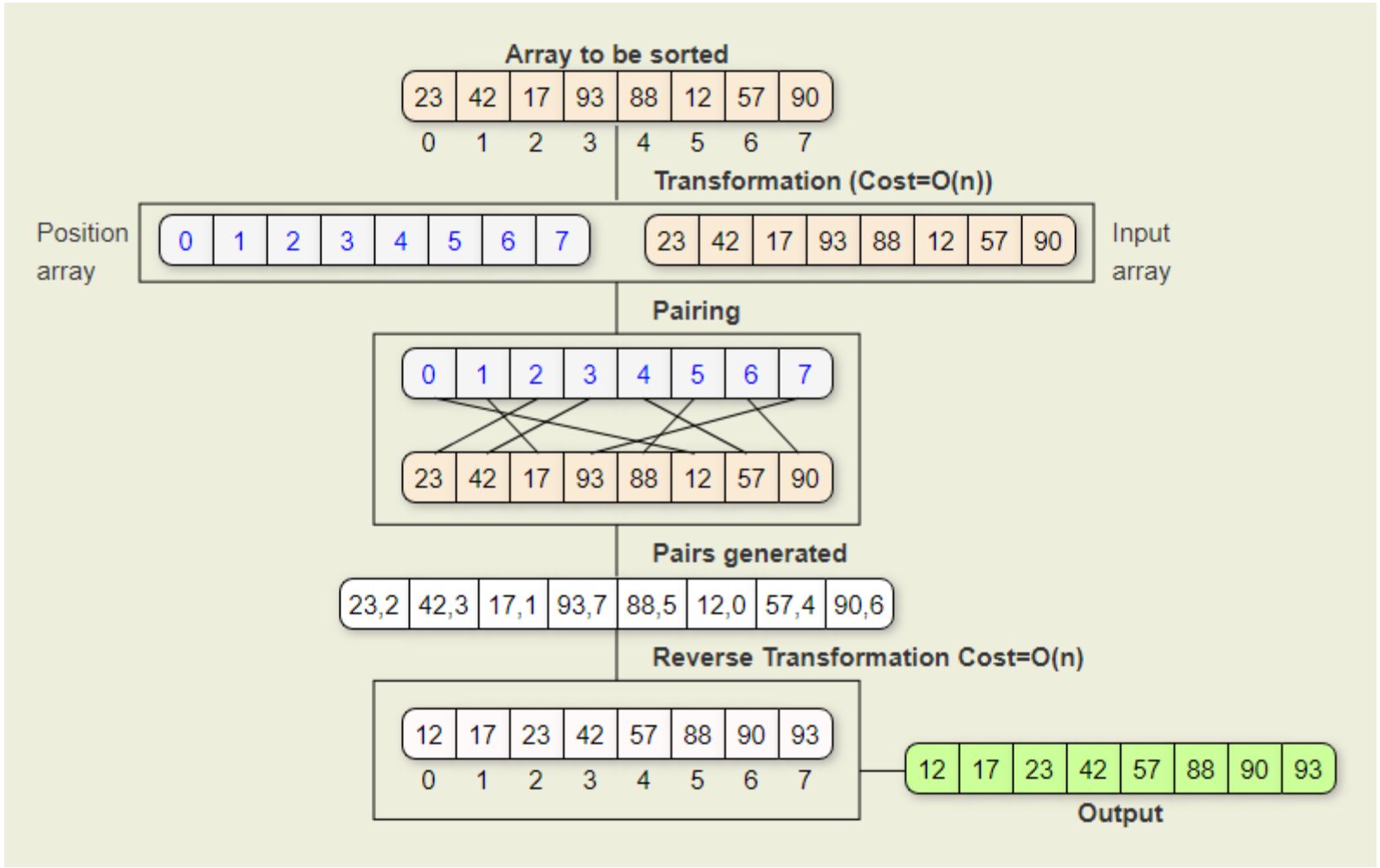
- Time Complexity?

Sorting of a given array by reducing it to Pairing problem.




Array to be sorted
23 42 17 93 88 12 57 90
0 1 2 3 4 5 6 7
Transformation (Cost=O(n))
Position array
0 1 2 3 4 5 6 7
23 42 17 93 88 12 57 90
Input array

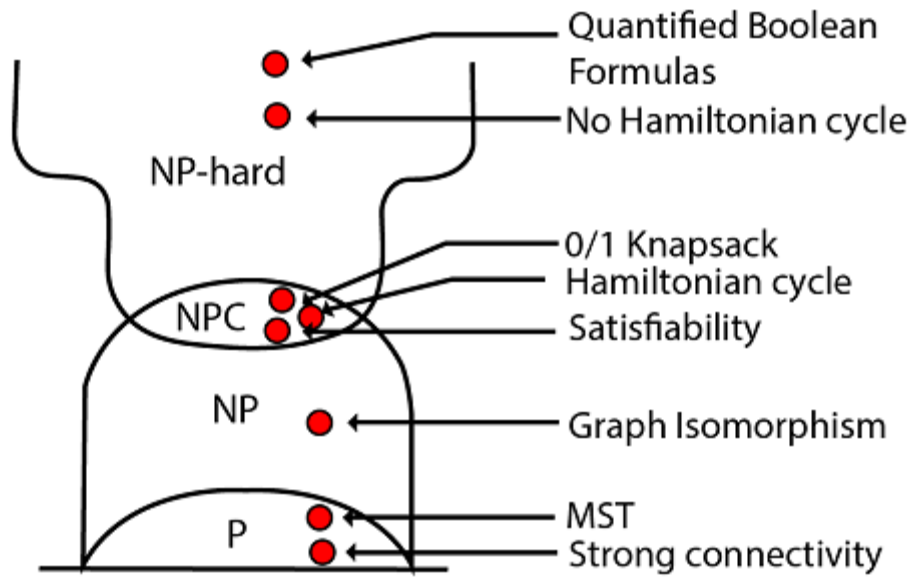

Array to be sorted
23 42 17 93 88 12 57 90
0 1 2 3 4 5 6 7
Transformation (Cost=O(n))
Position array
0 1 2 3 4 5 6 7
23 42 17 93 88 12 57 90
Input array
Pairing
0 1 2 3 4 5 6 7
23 42 17 93 88 12 57 90
Pairs generated
23,2 42,3 17,1 93,7 88,5 12,0 57,4 90,6


Array to be sorted
23 42 17 93 88 12 57 90
0 1 2 3 4 5 6 7
Transformation (Cost=O(n))
Position array
0 1 2 3 4 5 6 7
23 42 17 93 88 12 57 90
Input array
Pairing
0 1 2 3 4 5 6 7
23 42 17 93 88 12 57 90
Pairs generated
23,2 42,3 17,1 93,7 88,5 12,0 57,4 90,6
Reverse Transformation Cost=O(n)
12 17 23 42 57 88 90 93
0 1 2 3 4 5 6 7
12 17 23 42 57 88 90 93
Output

NP-Hard

- A problem X is said to be NP-Hard if all problems in NP are reducible(efficiently) to X or in other words a very difficult problem .

- If R is polynomial-time reducible to Q, we denote this $R <=_p Q$
- If all problems $R \in NP$ are polynomial-time reducible to Q, then Q is NP-Hard.
- If $R <=_p Q$ and R is NP-Hard, Q is also NP-Hard


Quantified Boolean
Formulas
No Hamiltonian cycle
NP-hard
0/1 Knapsack
Hamiltonian cycle
NPC
Satisfiability
NP
Graph Isomorphism
MST
P
Strong connectivity

Unit 2

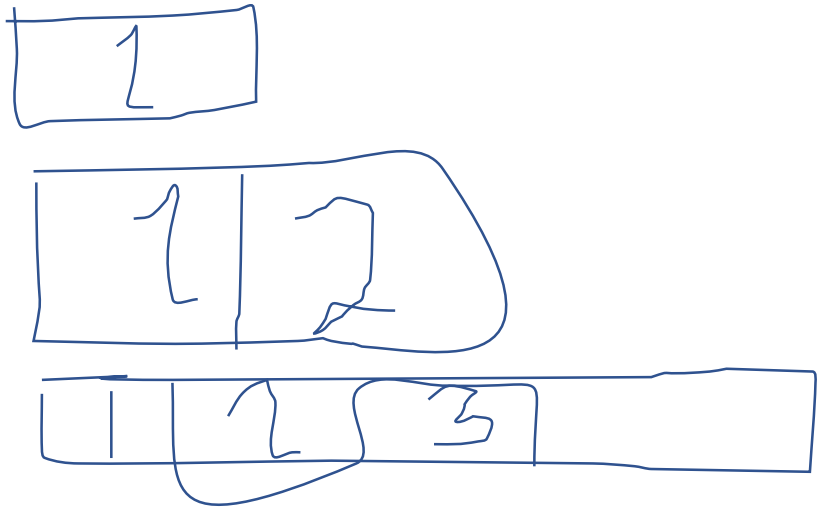
Amortized analysis

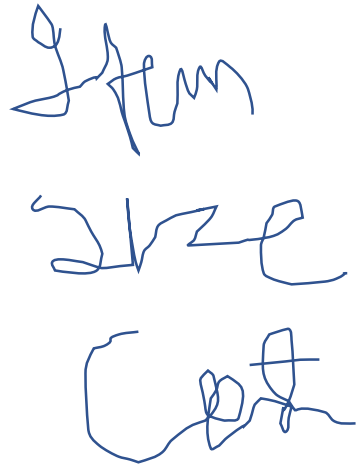
- In an amortized analysis, we average the time required to perform a sequence of data structure operations over all the operations performed.
- Using an amortized analysis, we can show that the average cost of an operation is small, even though a single operation within the sequence might be expensive.
- Used to analysed time complexities of hash tables, splay trees, disjoint sets.
- Techniques used
- -Aggregate method
- -Accounting method
- -Potential method

Aggregate analysis

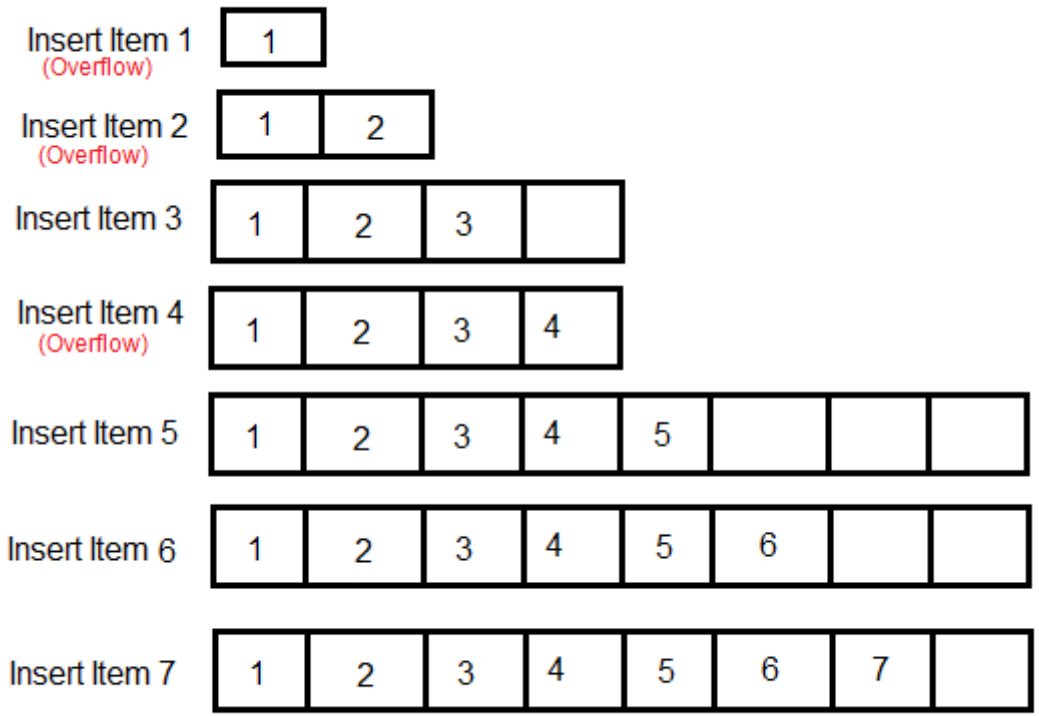
Hash table insertion

- Increase the size of table whenever it becomes full
- -Allocate memory for a large table of size double the old table
- -copy the contents of old table to new table
- -Free the old table

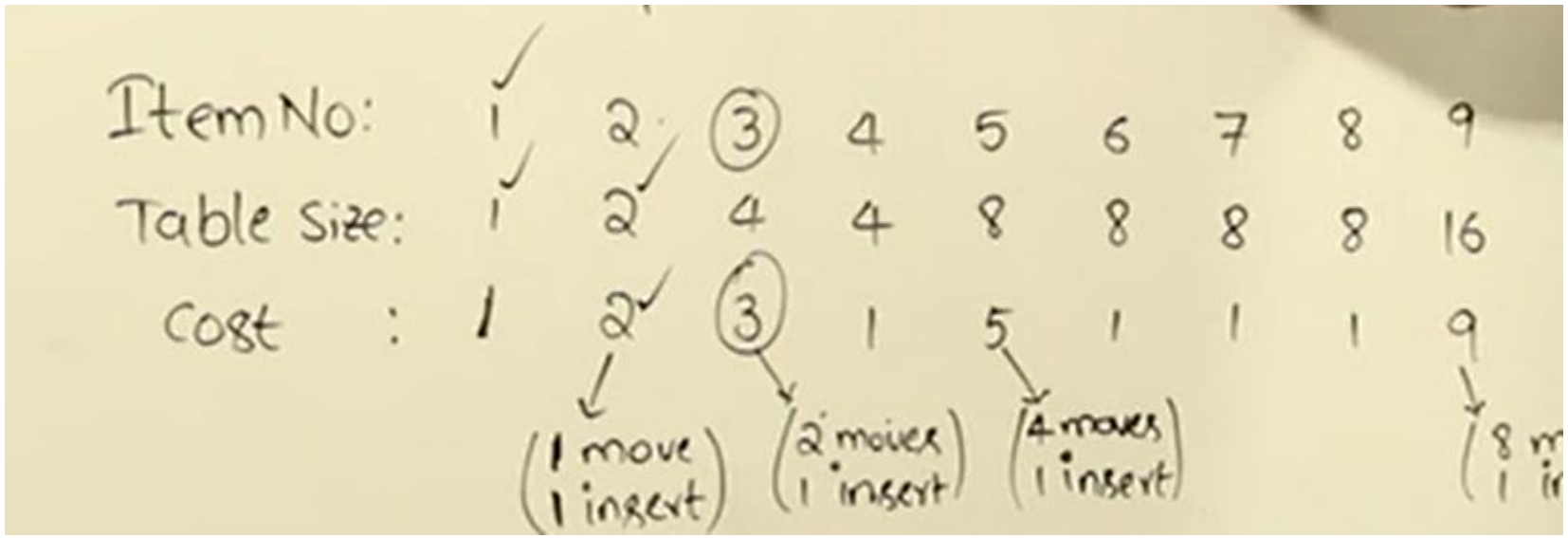




Initially table is empty and size is 0



Next overflow would happen when we insert 9, table size would become 16


Item No: 1 2 3 4 5 6 7 8 9
Table Size: 1 2 4 4 8 8 8 8 16
Cost : 1 2 3 1 5 1 1 1 9
(1 move 1 insert)
(2 moves 1 insert)
(4 moves 1 insert)

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + ...)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

- n*O(n)
- Amortized cost = (1+2+3+1+5+1+1+1+9…..)/n
- =(1+1+1+1….) + (1+2+4+8….)/n
- =(n+2n)/n
- Amortized cost = O(1)

Greedy

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Examples of Greedy algorithm

- -MST: Prim’s and Kruskal’s
- -SSSP: Disjkstra
- -BFS and DFS

Fractional/Greedy Knapsack Problem

- Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- Portfolio optimization

There are basically three approaches to solve the problem:

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- The third approach is to calculate the ratio of profit/weight.

- **Consider the below example:**
- Objects: 1 2 3 4 5 6 7
- Profit (P): 5 10 15 7 8 9 4
- Weight(w): 1 3 5 4 1 3 2
- W (Weight of the knapsack): 15
- n (no of items): 7

- **First approach:**

Object	Profit	Weight	Remaining weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
4	7 * ¾ = 5.25	3	3 - 3 = 0

The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25

Second approach:

The second approach is to select the item based on the minimum weight.

Object	Profit	Weight	Remaining weight
1	5	1	15 - 1 = 14
5	7	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	15 * 1/5 = 3	1	1 - 1 = 0

In this case, the total profit would be equal to (5 + 7 + 4 + 10 + 9 + 7 + 3) = 46

Third approach:

In the third approach, we will calculate the ratio of profit/weight.

Objects: 1 2 3 4 5 6 7

Profit (P): 5 10 15 7 8 9 4

Weight(w): 1 3 5 4 1 3 2

In this case, we first calculate the profit/weight ratio.

Object 1: 5/1 = 5

Object 2: 10/3 = 3. 33

Object 3: 15/5 = 3

Object 4: 7/4 = 1.7

Object 5: 8/1 = 8

Object 6: 9/3 = 3

Object 7: 4/2 = 2

P:w: 5 3.3 3 1.7 8 3 2

In this approach, we will select the objects based on the maximum profit/weight ratio. Since the P/W of object 5 is maximum so we select object 5.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0

As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leqslant x_i \leqslant 1$$

The i^{th} item contributes the weight $x_i . w_i$ to the total weight in the knapsack and profit $x_i . p_i$ to the total profit.

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n} (x_i . pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i . wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i . wi) = W$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1} \leq \frac{p_i}{w_i}$. Here, x is an array to store the fraction of items.

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

Analysis

- If the provided items are already sorted into a decreasing order of pi/wi, then the whileloop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Example

Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Job Sequencing with Deadline

Problem Statement

- In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Points to Remember for Job Sequencing with Deadlines

- Each job has deadline d_i & it can process the job within its deadline; only one job can be processed at a time.
- Only one CPU is available for processing all jobs.
- CPU can take only one unit at a time for processing any job.
- All jobs arrived at the same time.

Greedy Algorithm-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

- Sort all the given jobs in decreasing order of their profit.

Step-02:

- Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:

- Pick up the jobs one by one.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Problem-

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Step-01:

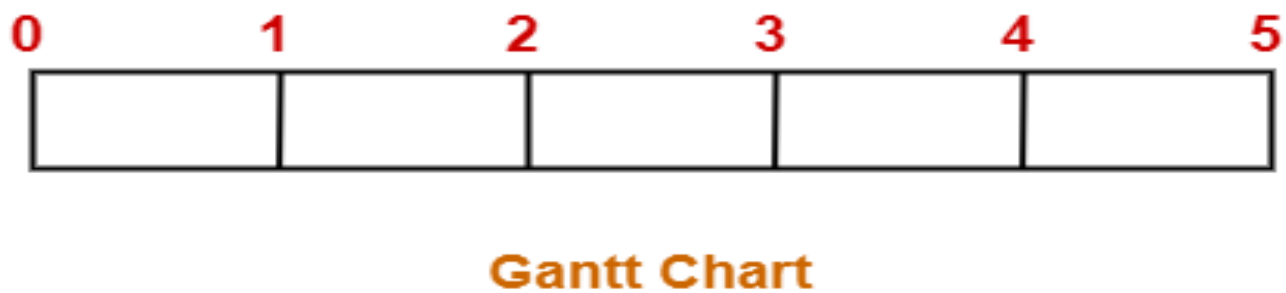
Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



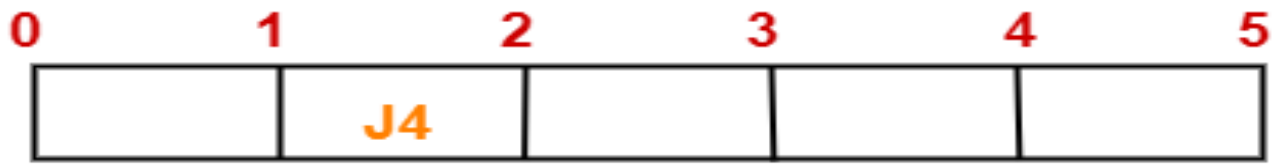
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

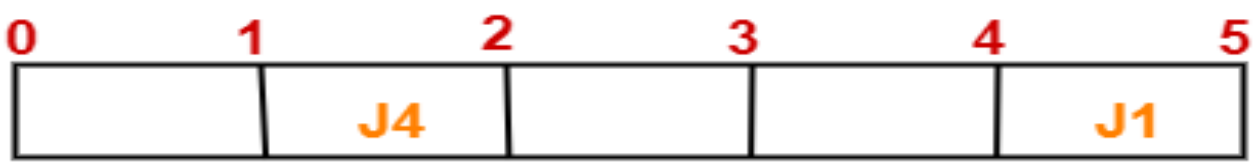
Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Step-06:

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

Part-01:

The optimal schedule is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Part-02:

- All the jobs are not completed in optimal schedule.
- This is because job J6 could not be completed within its deadline.

Part-03:

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200

= 990 units

Algorithm

- Sort the jobs based on decreasing order of profit.
- Iterate through the jobs and perform the following:
  - Choose a **Slot i** if:
    - **Slot i** isn't previously selected.
    - **I < deadline**
    - **I** is maximum
  - If no such slot exists, ignore the job and continue.

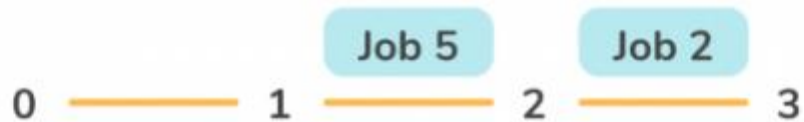
Job ID	1	2	3	4	5
Deadline	2	3	2	1	3
Profit	20	38	16	10	30

in decreasing order of profits:

Job ID	2	5	1	3	4
Deadline	3	3	2	2	1
Profit	38	30	20	16	10

1. The last empty slot available for Job 2 before deadline is slot 2-3

2. The last empty slot available for Job 5 before deadline is slot 1-2



2. The last empty slot available for Job 1 before deadline is slot 0-1

All the slots are full. So, the sequence of jobs is : 1 5 2

Huffman Coding-

- Huffman Coding is a famous Greedy Algorithm.
- It is used for the lossless compression of data.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as **Huffman Encoding**.

Huffman Tree-

The steps involved in the construction of Huffman Tree are as follows-

Step-01:

- Create a leaf node for each character of the text.
- Leaf node of a character contains the occurring frequency of that character.

Step-02:

- Arrange all the nodes in increasing order of their frequency value.

Step-03:

Considering the first two nodes having minimum frequency,

- Create a new internal node.
- The frequency of this new node is the sum of frequency of those two nodes.
- Make the first node as a left child and the other node as a right child of the newly created node.

Formula-01:

$$\text{Average code length per character} = \frac{\Sigma\ (\text{frequency}_i \times \text{code length}_i)}{\Sigma\ \text{frequency}_i}$$

$$= \Sigma\ (\text{probability}_i \times \text{code length}_i)$$

Formula-02:

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= $\sum$ (frequency_i x Code length_i)

Step-04:

- Keep repeating Step-02 and Step-03 until all the nodes form a single tree.
- The tree finally obtained is the desired Huffman Tree.

Time Complexity-

The time complexity analysis of Huffman Coding is as follows-

- extractMin( ) is called 2 x (n-1) times if there are n nodes.
- As extractMin( ) calls minHeapify( ), it takes O(logn) time.

Thus, Overall time complexity of Huffman Coding becomes **O(nlogn)**.

Here, n is the number of unique characters in the given text.

Problem-

A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-

1. Huffman Code for each character
2. Average code length
3. Length of Huffman encoded message (in bits)

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

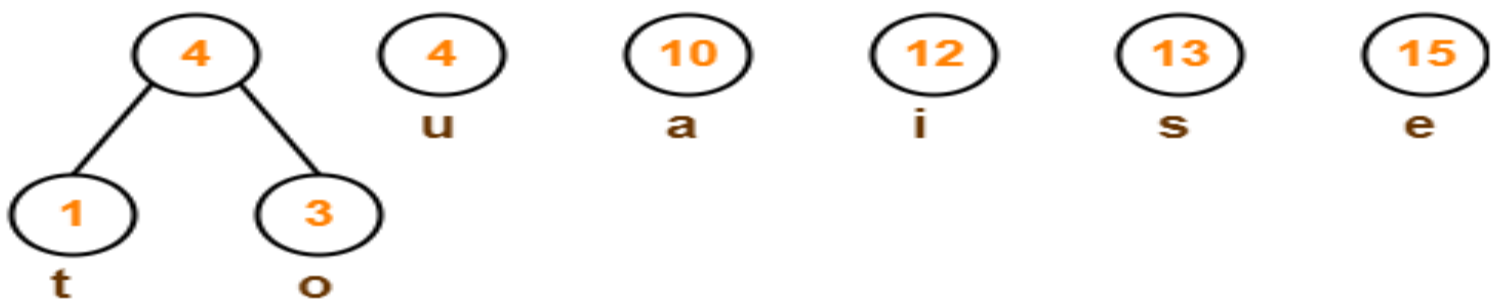
Solution-

First let us construct the Huffman Tree.

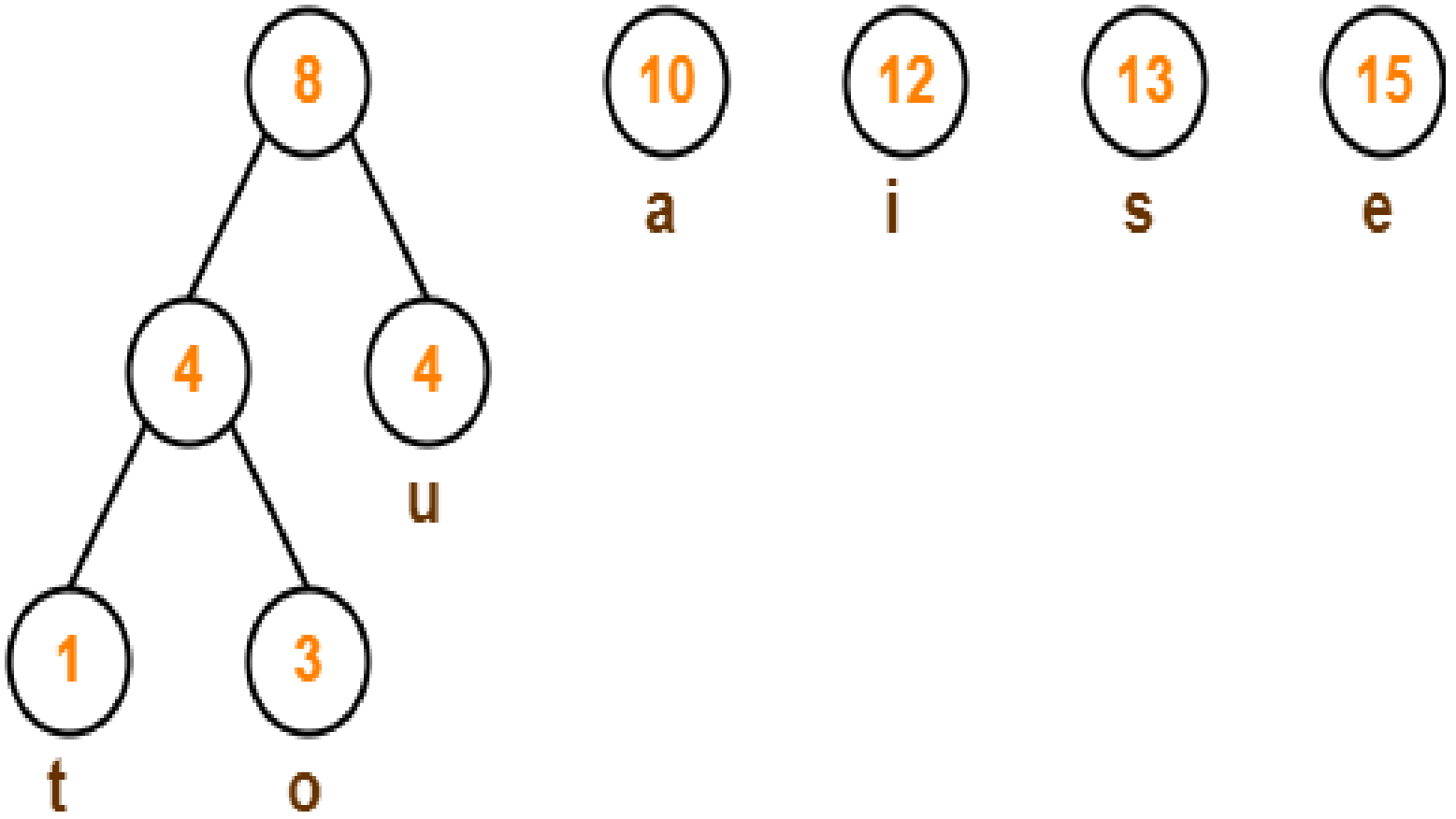
Huffman Tree is constructed in the following steps-

Step-01:

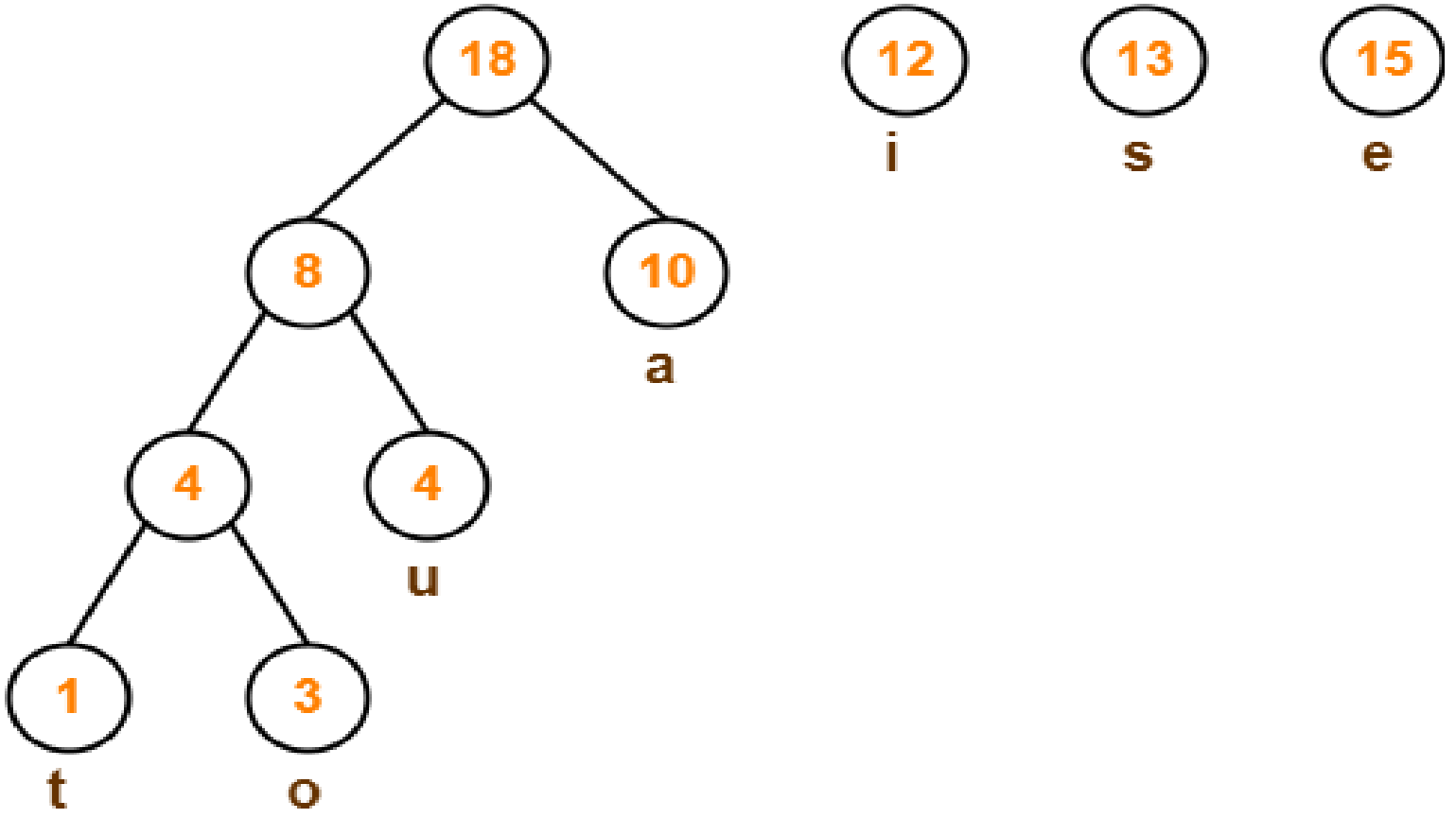
Step-02:



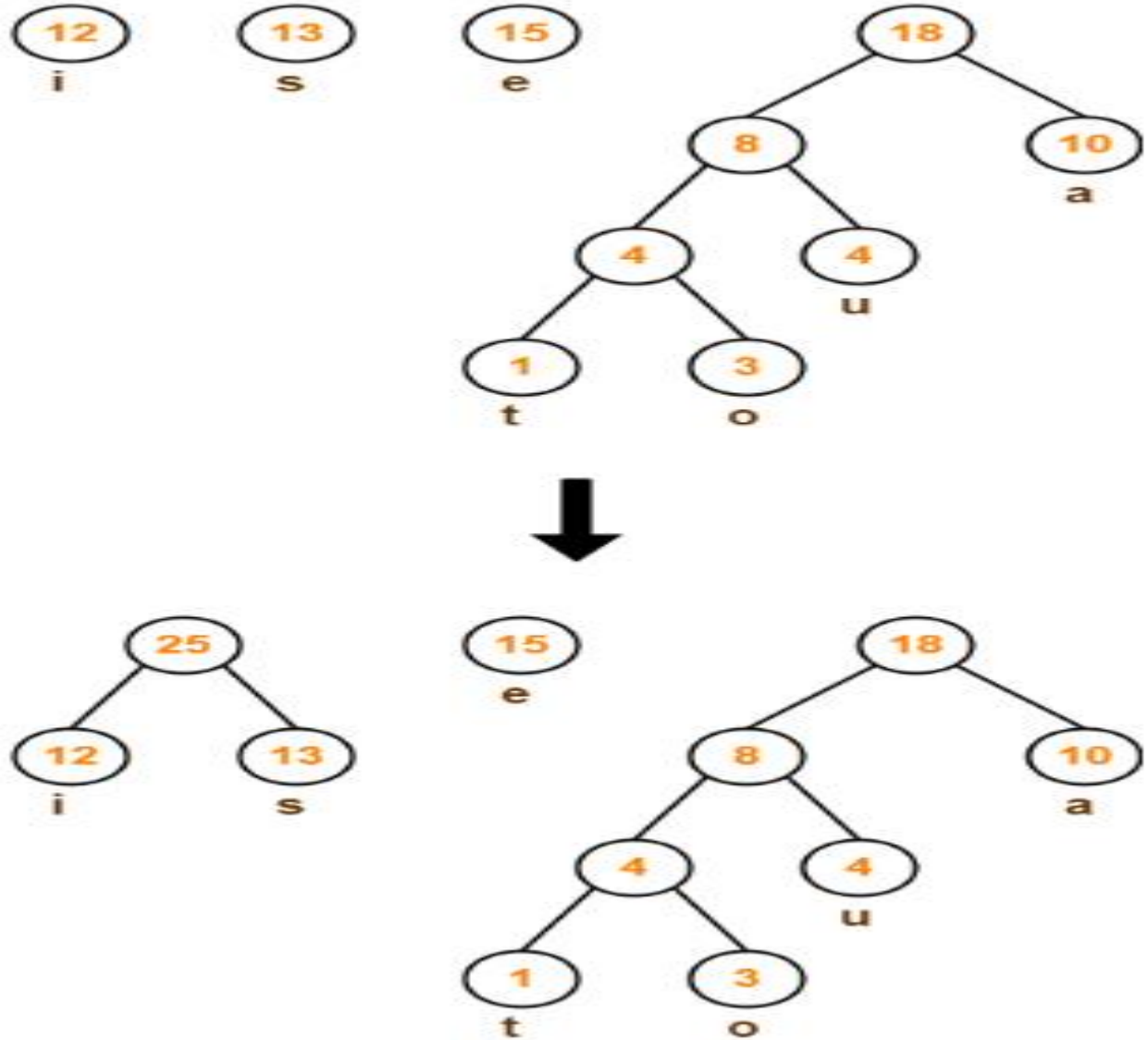
Step-03:



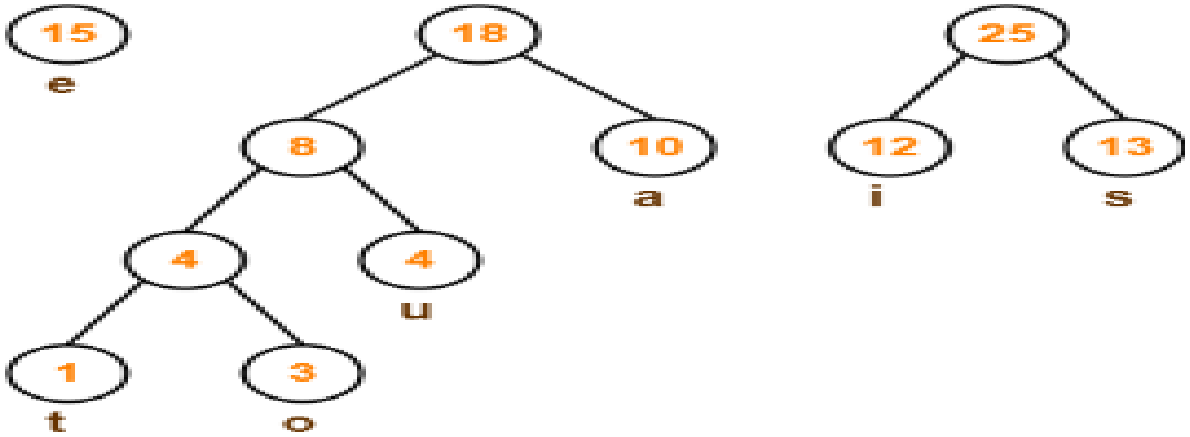
Step-04:

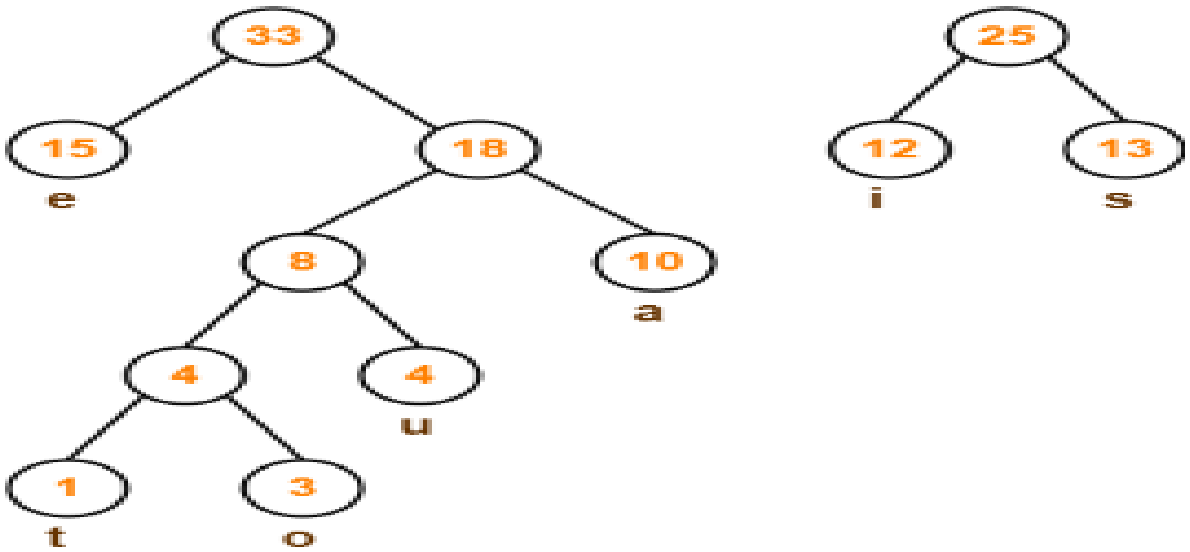


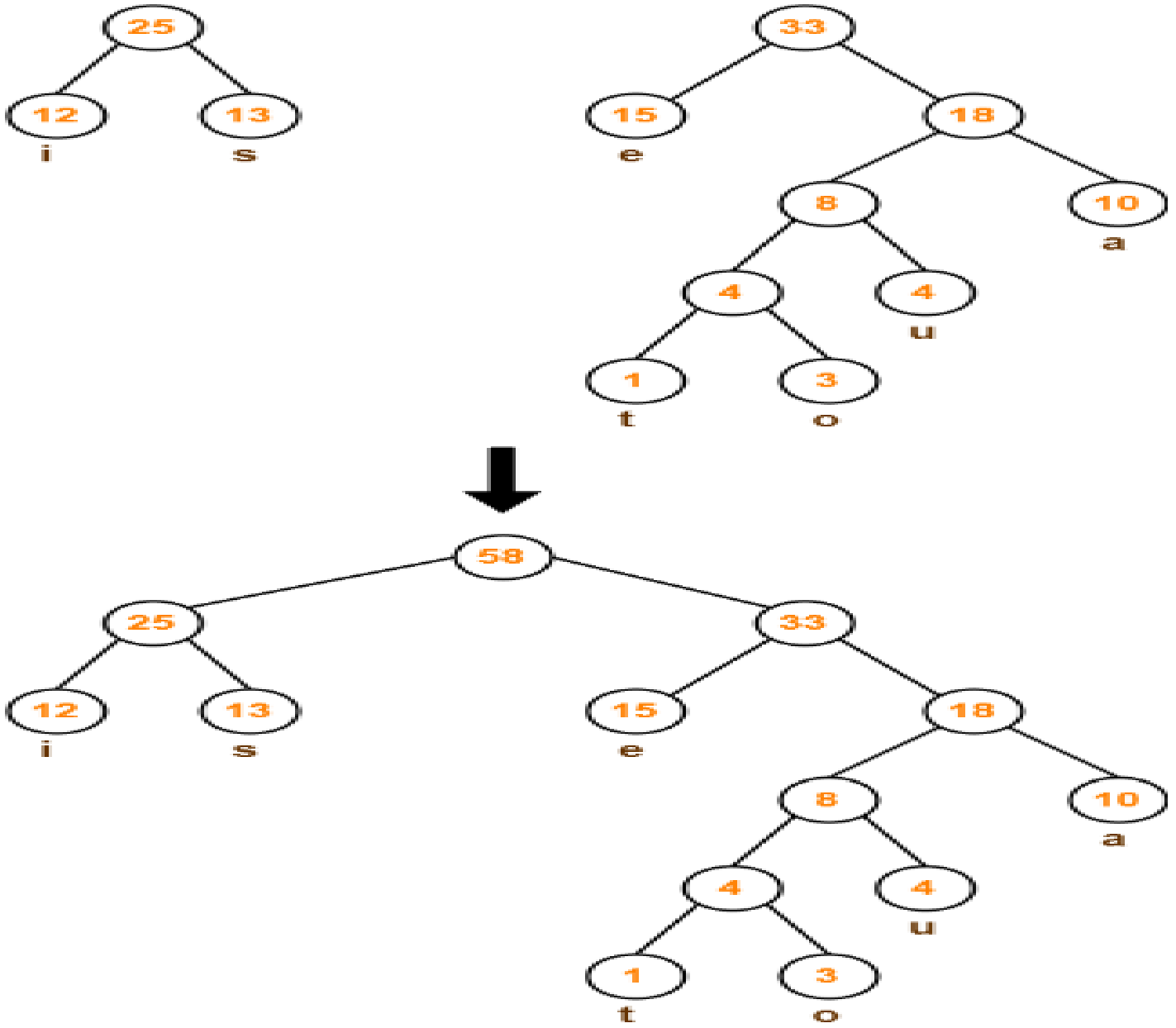
Step-05:



Step-06:


15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s


33
15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
13
i
s


25
12
i
13
s
33
15
e
18
8
10
a
4
4
u
1
t
3
o
58
25
12
i
13
s
33
15
e
18
8
10
a
4
4
u
1
t
3
o

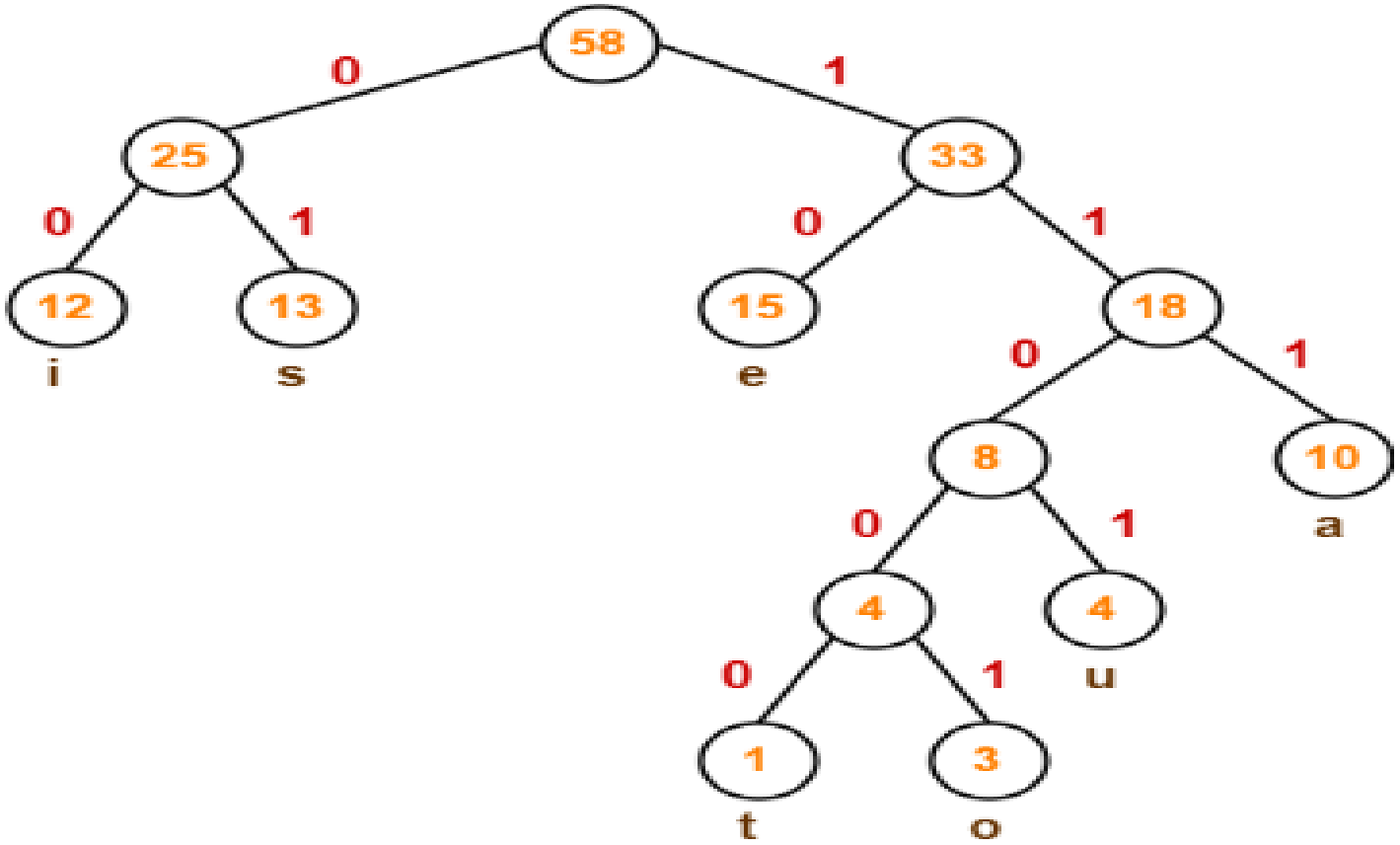
Now,

- We assign weight to all the edges of the constructed Huffman Tree.
- Let us assign weight '0' to the left edges and weight '1' to the right edges.

Rule

- If you assign weight '0' to the left edges, then assign weight '1' to the right edges.
- If you assign weight '1' to the left edges, then assign weight '0' to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

1. Huffman Code For Characters-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

2. Average Code Length-

Using formula-01, we have-

Average code length

$= \sum (\text{frequency}_i \times \text{code length}_i) / \sum (\text{frequency}_i)$

$= \{ (10 \times 3) + (15 \times 2) + (12 \times 2) + (3 \times 5) + (4 \times 4) + (13 \times 2) + (1 \times 5) \} / (10 + 15 + 12 + 3 + 4 + 13 + 1)$

$= 2.52$

3. Length of Huffman Encoded Message-

Using formula-02, we have-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

$= 58 \times 2.52$

$= 146.16$

$\cong 147$ bits

Algorithm of Huffman Code

Huffman (C)

```
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Dynamic Programming

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.
- Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.
- The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

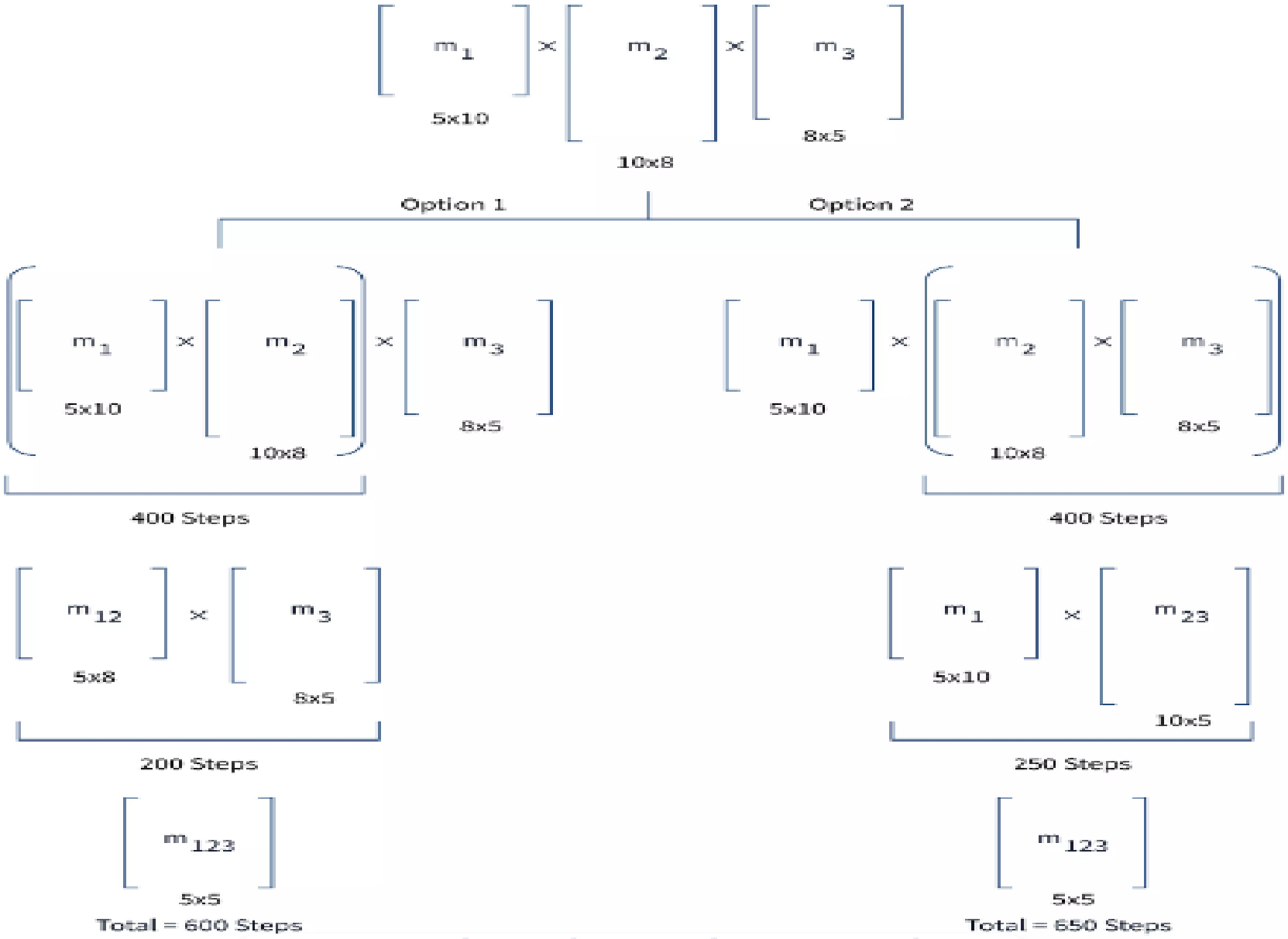
Matrix Chain Multiplication

- The number of multiplication steps required to multiply a chain of matrices of any arbitrary length *n* highly depends on the order in which they get multiplied. As multiplication of two numbers/integers is a costly operation, it is very much needed to find the optimal order of multiplying a chain of matrices to minimize the multiplication steps, which ultimately optimizes the time and resources required for the task.

Example of Matrix Chain Multiplication

- We have studied in our school days that matrix multiplications follow associative rule. The number of steps required in multiplying matrix $M1$ of dimension $x \times y$ and $M2$ of dimension $y \times z$ are $x \times y \times z$, and the resultant matrix $M12$ is of dimension $x \times z$
- Let's assume we have three matrices $M1, M2, M3$ of dimensions 5×10, 10×8, and 8×5 respectively and for some reason we are interested in multiplying all of them. There are two possible orders of multiplication –

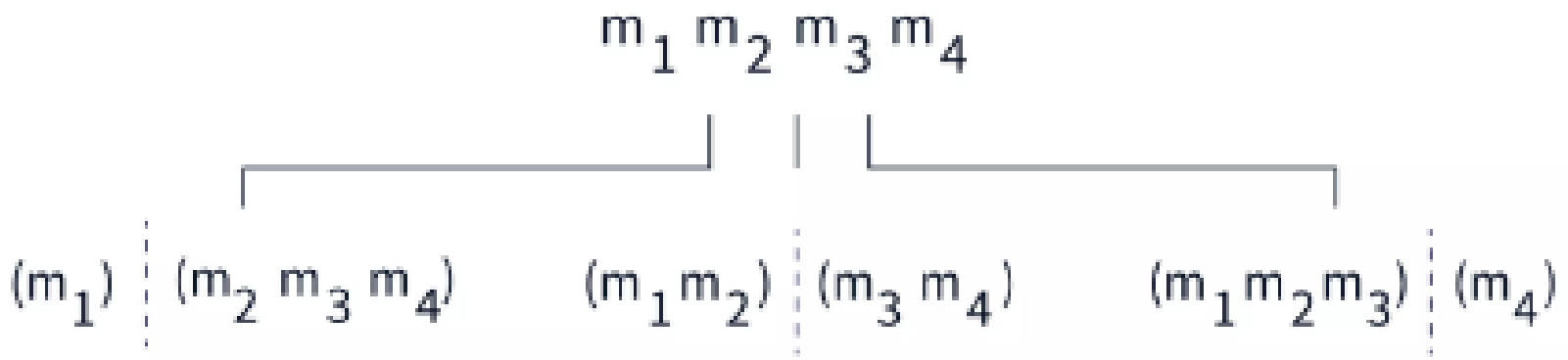
- $M1 \times (M2 \times M3)$
  - Steps required in $M2 \times M3$ will be $10 \times 8 \times 5 = 400$.
  - Dimensions of $M23$ will be 10×5.
  - Steps required in $M1 \times M23$ will be $5 \times 10 \times 5 = 250$.
  - Total Steps $= 400 + 250 = 650$
- $(M1 \times M2) \times M3$
  - Steps required in $M1 \times M2$ will be $5 \times 10 \times 8 = 400$.
  - Dimensions of $M12$ will be 5×8.
  - Steps required in $M12 \times M3$ will be $5 \times 8 \times 5 = 200$.
  - Total Steps $= 400 + 200 = 600$


m1
5x10
m2
10x8
m3
8x5
Option 1
Option 2
400 Steps
400 Steps
m12
5x8
200 Steps
m123
5x5
Total = 600 Steps
m23
10x5
250 Steps
Total = 650 Steps

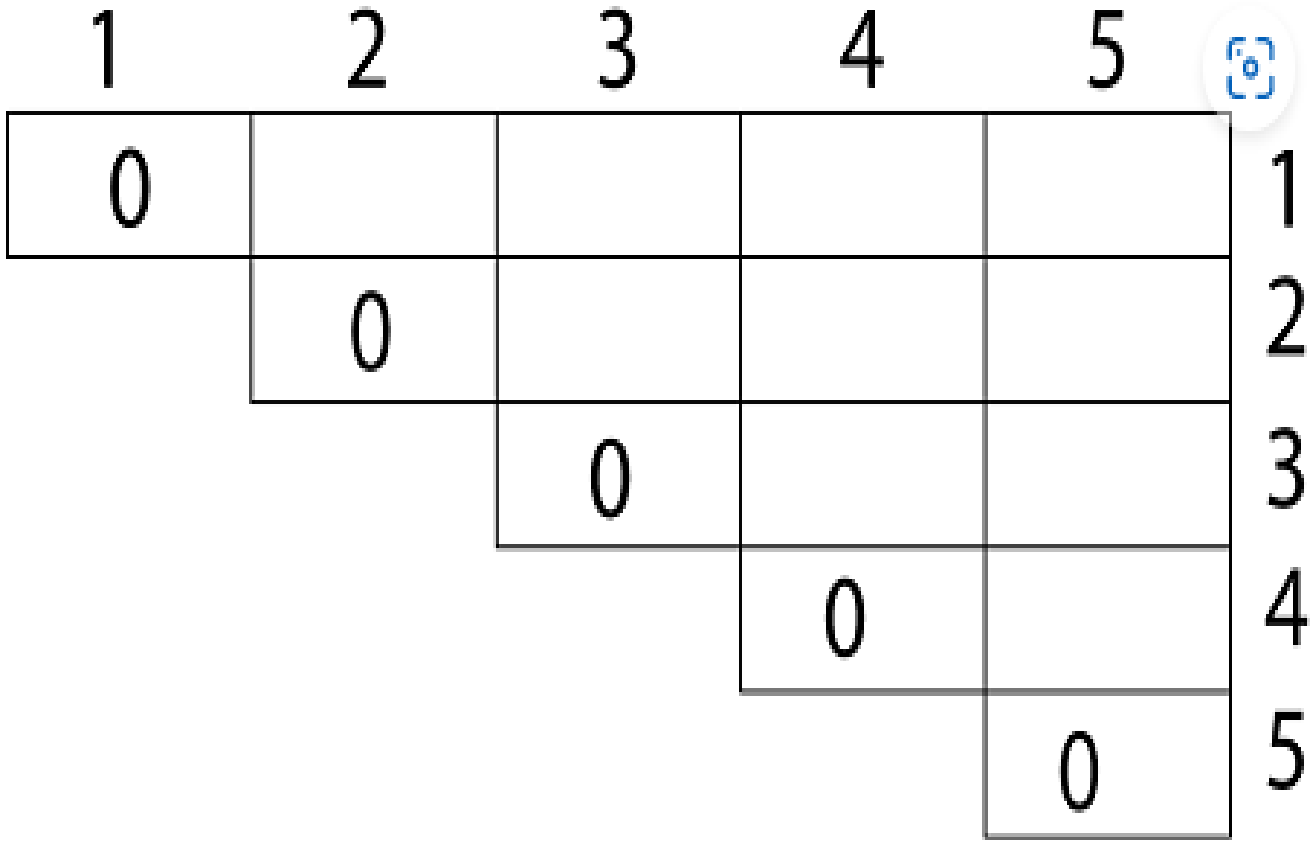
Naive Recursive Approach

The idea is very simple, placing parentheses at every possible place. For example, for a matrix chain $M_1 M_2 M_3 M_4$ we can have place parentheses like -

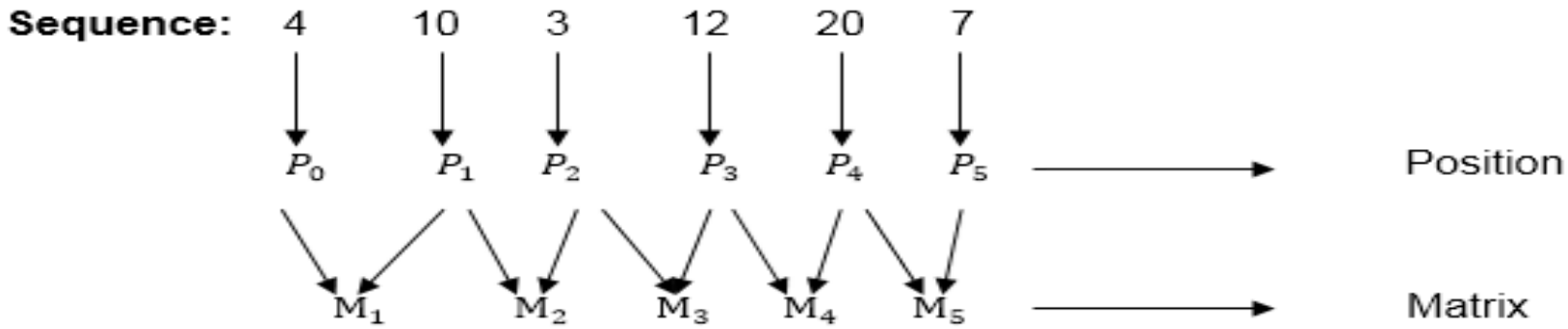
- $(M_1)(M_2 M_3 M_4)$
- $(M_1 M_2)(M_3 M_4)$
- $(M_1 M_2 M_3)(M_4)$



Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], $0 \leq i, j \leq 5$. We know M [i, i] = 0 for all i.



Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})

On the basis of sequence, we make a formula

For M_i ⟶ p [i] as column

p [i-1] as row

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. m (1,2) = m_1 x m_2
   = 4 x 10 x 10 x 3
   = 4 x 10 x 3 = 120

2. m (2, 3) = m_2 x m_3
   = 10 x 3 x 3 x 12
   = 10 x 3 x 12 = 360

3. m (3, 4) = m_3 x m_4
   = 3 x 12 x 12 x 20
   = 3 x 12 x 20 = 720

4. m (4,5) = m_4 x m_5
   = 12 x 20 x 20 x 7
   = 12 x 20 x 7 = 1680

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Now product of 3 matrices:

```
M [1, 3] = M1 M2 M3
```

1. There are two cases by which we can solve this multiplication: (M_1 x M_2) + M_3, M_1+ (M_2x M_3)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[1,3] = \min \begin{cases} M\,[1,2] + M\,[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M\,[1,1] + M\,[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M_1 x M_2) + M_3 this combination is chosen for the output making.

```
M [2, 4] = M2 M3 M4
```

1. There are two cases by which we can solve this multiplication: (M_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

$$M\,[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and $M_2+(M_3 \times M_4)$ this combination is chosen for the output making.

```
M [3, 5] = M3  M4  M5
```

1. There are two cases by which we can solve this multiplication: $(M_3 \times M_4) + M_5$, $M_3 + (M_4 \times M_5)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

```
M [3, 5] = 1140
```

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and $(M_3 \times M_4) + M_5$ this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:

```
M [1, 4] = M1  M2 M3 M4
```

There are three cases by which we can solve this multiplication:

1. (M_1 x M_2 x M_3) M_4
2. M_1 x(M_2 x M_3 x M_4)
3. (M_1 xM_2) x (M_3 x M_4)

After solving these cases we choose the case in which minimum output is there

$$M\,[1,4] = \min \begin{Bmatrix} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{Bmatrix}$$

M [1, 4] =1080

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M_1 xM_2) x (M_3 x M_4) combination is taken out in output making,

```
M [2, 5] = M2 M3 M4 M5
```

There are three cases by which we can solve this multiplication:

1. (M_2 x M_3 x M_4)x M_5
2. M_2 x(M_3 x M_4 x M_5)
3. (M_2 x M_3)x (M_4 x M_5)

After solving these cases we choose the case in which minimum output is there

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

```
M [2, 5] = 1350
```

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M_2 x(M_3 x M_4 xM_5)combination is taken out in output making.

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

→

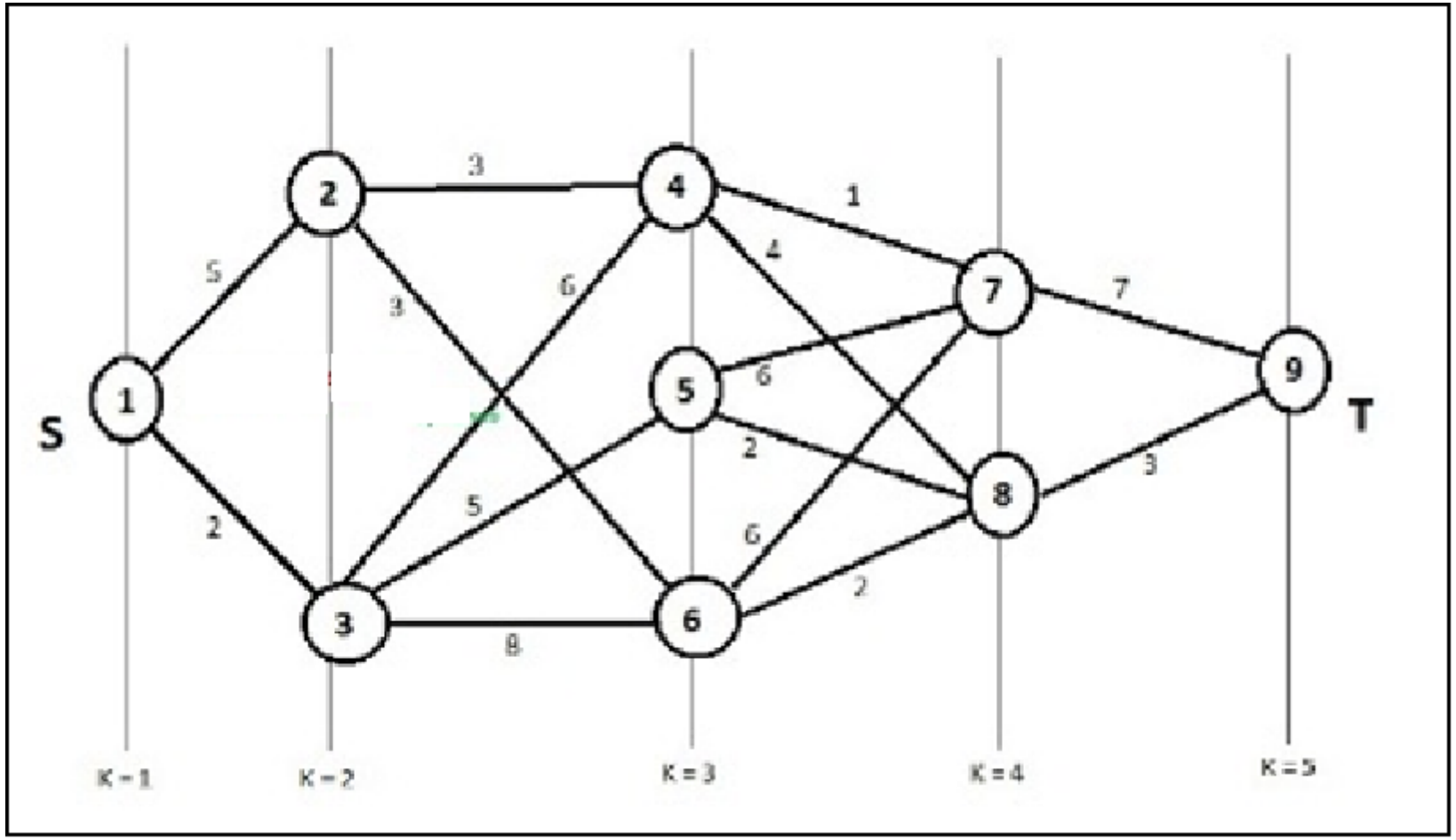
1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

```
int B(int p[], int i, int j)
{
    if(i == j)
        return 0
    int min = INT_MAX
    int count
    for (int k = i to k < j) {
        count = B(p, i, k) +   B(p, k + 1, j) +  p[i - 1] * p[k] * p[j]
       if (count < min)
            min = count
    }
    return min
```

Multistage Graph

- A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets ***S* = {s_1,s_2,…,s_k}** such that edge *(u, v)* is in E, then *u* $\in$ s_i and *v* $\in$ s_{1+1} for some subsets in the partition and $|s_1| = |s_k| = 1$.
- The vertex ***s* $\in$ s_1** is called the **source** and the vertex ***t* $\in$ s_k** is called **sink**.
- ***G*** is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

$$cost(i,j)=min\{c(j,l)+cost(i+1,l)\}$$


S
T
1
2
3
4
5
6
7
8
9
5
3
2
3
6
1
4
6
7
6
5
2
6
2
3
8
K=1
K=2
K=3
K=4
K=5

- *Cost(5, 9) = 0*
- *Cost(4, 7) = min {c(7, 9) + Cost(5, 9)} = 7 + 0 = 7*

- *Cost(4, 8) = min {c(8, 9) + Cost(5, 9)} = 3 + 0 = 3*

- $Cost(3, 4) = min \{c(4, 7) + Cost(4, 7), c(4, 8) + Cost(4, 8)\}$
  $= min\{1 + 7, 4 + 3\} = 7$
- $Cost(3, 5) = min \{c(5, 7) + Cost(4, 7), c(5, 8) + Cost(4, 8)\}$
  $= min\{6 + 7, 2 + 3\} = 5$
- $Cost(3, 6) = min \{c(6, 7) + Cost(4, 7), c(6, 8) + Cost(4, 8)\}$
  $= min\{6 + 7, 2 + 3\} = 5$

- $Cost(2, 2) = min \{c(2, 4) + Cost(3, 4), c(2, 6) + Cost(3, 6)\}$

  $= min\{3 + 7, 3 + 5\} = 8$

- $Cost(2, 3) = min\{c(3, 4) + Cost(3, 4), c(3, 5) + Cost(3, 5), c(3, 6) + Cost(3, 6)\}$

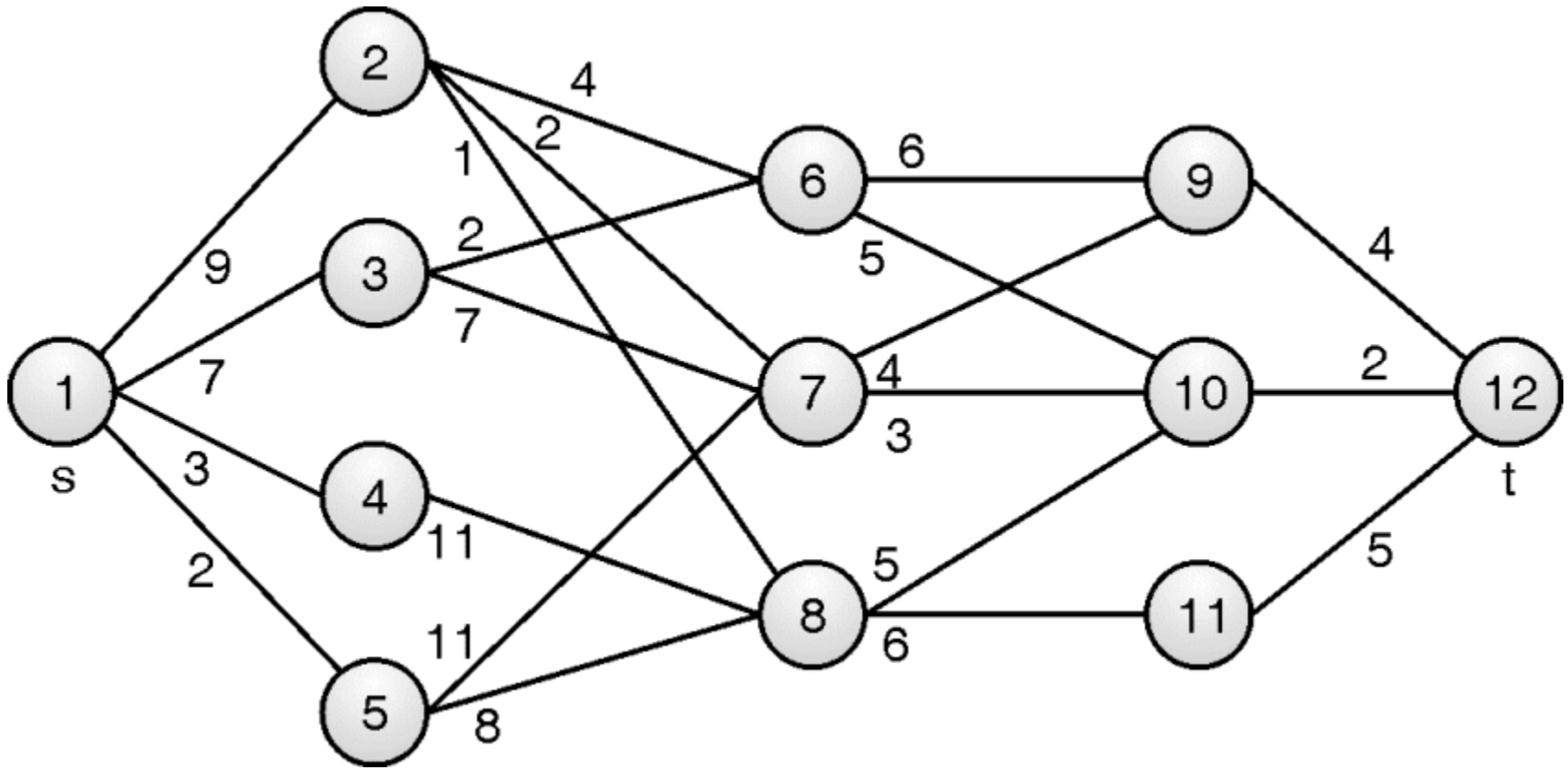
  $= min\{6 + 7, 5 + 5, 8 + 5\}= 10$

- *Cost (1, 1) = {c(1, 2) + Cost(2, 2), c(1, 3) + Cost(2, 3)}*

  = min{5 + 8, 2 + 10} = 12

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**

- Algorithm MULTI_STAGE(G, k, n, p)
- // Input:
- k: Number of stages in graph G = (V, E)
- c[i, j]:Cost of edge (i, j)
- // Output: p[1:k]:Minimum cost path
- cost[n] ← 0
- for j ← n – 1 to 1 do
- //Let r be a vertex such that (j, r) in E and c[j, r] + cost[r] is minimum
- cost[j] ← c[j, r] + cost[r]
- π[j] ← r
- end
- //Find minimum cost path
- p[1] ← 1
- p[k] ← n
- for j ← 2 to k - 1 do
- p[j] ← π[p[j - 1]]
- end


1
s
2
3
4
5
6
7
8
9
10
11
12
t
9
7
3
2
4
1
2
7
4
11
11
8
4
6
5
3
5
6
2
4
5

0/1 knapsack problem

- The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Consider the problem having weights and profits are:

- Weights: {3, 4, 6, 5}
- Profits: {2, 3, 1, 4}
- The weight of the knapsack is 8 kg
- The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

using the Dynamic programming approach

- The first row and the first column would be 0 as there is no item for w=0

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

- **When i=1, W=1**

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0							
2	0								
3	0								
4	0								

- **When i = 1, W = 2**

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0						
2	0								
3	0								
4	0								

- **When i=1, W=3**

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2					
2	0								
3	0								
4	0								

When i=1, W = 4

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2				
2	0								
3	0								
4	0								

- **When i=1, W = 5**

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2			
2	0								
3	0								
4	0								

When i =1, W=6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2		
2	0								
3	0								
4	0								

When i=1, W = 7

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	
2	0								
3	0								
4	0								

When i =1, W =8

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0								
3	0								
4	0								

When i =2, W = 1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0							
3	0								
4	0								

When i =2, W = 2

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0						
3	0								
4	0								

When i =2, W = 3

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2					
3	0								
4	0								

When i =2, W = 4

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3				
3	0								
4	0								

When i = 2, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3			
3	0								
4	0								

When i = 2, W = 6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3		
3	0								
4	0								

When i = 2, W = 7

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	0	3	3	3	5	
3	0								
4	0								

When i = 2, W = 8

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0								
4	0								

When i = 3, W = 1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0							
4	0								

When i = 3, W = 2

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0						
4	0								

When i = 3, W = 3

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2					
4	0								

When i = 3, W = 4

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3				
4	0								

When i = 3, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3			
4	0								

When i =3, W = 6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3		
4	0								

When i =3, W = 7

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	
4	0								

When i = 3, W = 8

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0								

When i = 4, W = 1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0							

When i = 4, W = 2

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0						

When i = 4, W = 3

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2					

When i = 4, W = 4

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3				

When i = 4, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3			

When i = 4, W = 6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	4	4		

When i = 4, W = 7

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	

When i = 4, W = 8

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	6

```
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   vector<vector<int> > K(n + 1, vector<int>(W + 1));

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1]
                          + K[i - 1][w - wt[i - 1]],
                      K[i - 1][w]);
         else
            K[i][w] = K[i - 1][w];
      }
   }
   return K[n][W];
```

Longest Common Subsequence

- Here longest means that the subsequence should be the biggest one. The common means that some of the characters are common between the two strings. The subsequence means that some of the characters are taken from the string that is written in increasing order to form a subsequence.
- **Let's understand the subsequence through an example.**
- Suppose we have a string 'w'.
- **W_1 = abcd**
- **The following are the subsequences that can be created from the above string:**
- ab
- bd
- ac
- ad
- acd
- bcd

- **Consider two strings:**
- X= a b a a b a
- Y= b a b b a b

For index i=1, j=1

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0					
b	0						
a	0						
a	0						
b	0						
a	0						

For index i=1, j=2

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1				
b	0						
a	0						
a	0						
b	0						
a	0						

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1			
b	0						
a	0						
a	0						
b	0						
a	0						

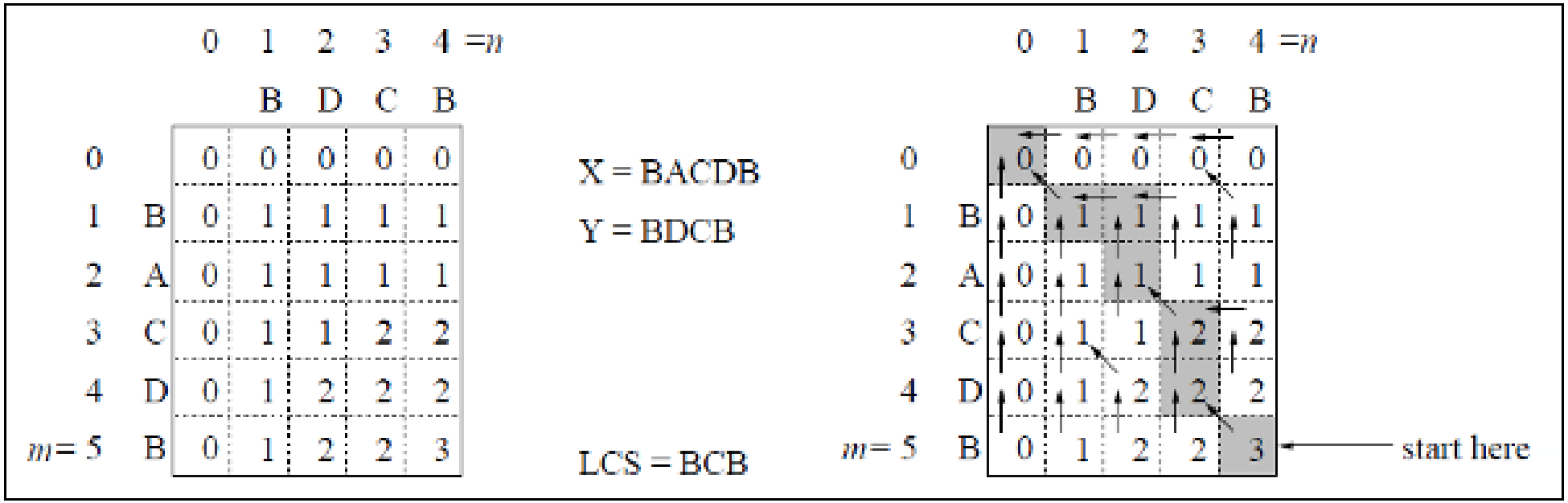
	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1	1		
b	0						
a	0						
a	0						
b	0						
a	0						

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1
b	0						
a	0						
a	0						
b	0						
a	0						

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1
b	0	1	1				
a	0						
a	0						
b	0						
a	0						

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1
b	0	1	1	2	2	2	2
a	0	1	2	2	2	3	3
a	0	1	2	2	2	3	3
b	0	1	1	3	3	3	4
a	0	1	2	2	2	4	4

- LCS = abab
- T.C. = O(n*m)


0 1 2 3 4 =n
B D C B
0 0 0 0 0 0
1 B 0 1 1 1 1
2 A 0 1 1 1 1
3 C 0 1 1 2 2
4 D 0 1 2 2 2
m= 5 B 0 1 2 2 3
X = BACDB
Y = BDCB
LCS = BCB
start here

String Matching Algorithms

- String matching operation is a core part in many text processing applications. The objective of this algorithm is to find pattern P from given text T. Typically |P|<< |T|. In the design of compilers and text editors, string matching operation is crucial. So locating P in T efficiently is very important.
- The problem is defined as follows: “Given some text string T[1….n] of size n, find all occurrences of pattern P[1…m] of size m in T.”

String Matching Algorithms can broadly be classified into two types of algorithms –

- Exact String Matching Algorithms
- Approximate String Matching Algorithms

Exact String Matching Algorithms:

Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence.

Approximate String Matching Algorithms:

Approximate String Matching Algorithms (also known as Fuzzy String Searching) searches for substrings of the input string.

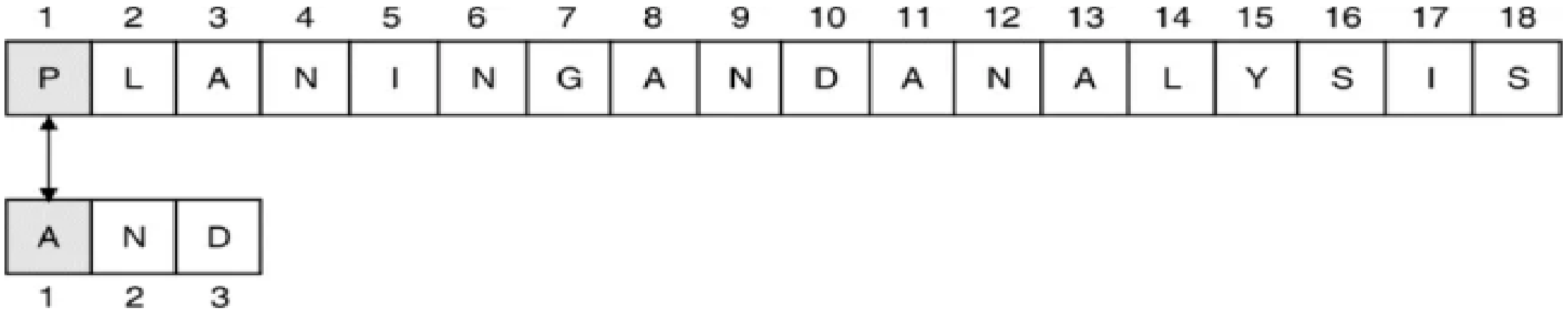
Applications of String Matching Algorithms:

- Plagiarism Detection
- Bioinformatics and DNA Sequencing
- Digital Forensics
- Spelling Checker
- Spam filters
- Search engines or content search in large databases

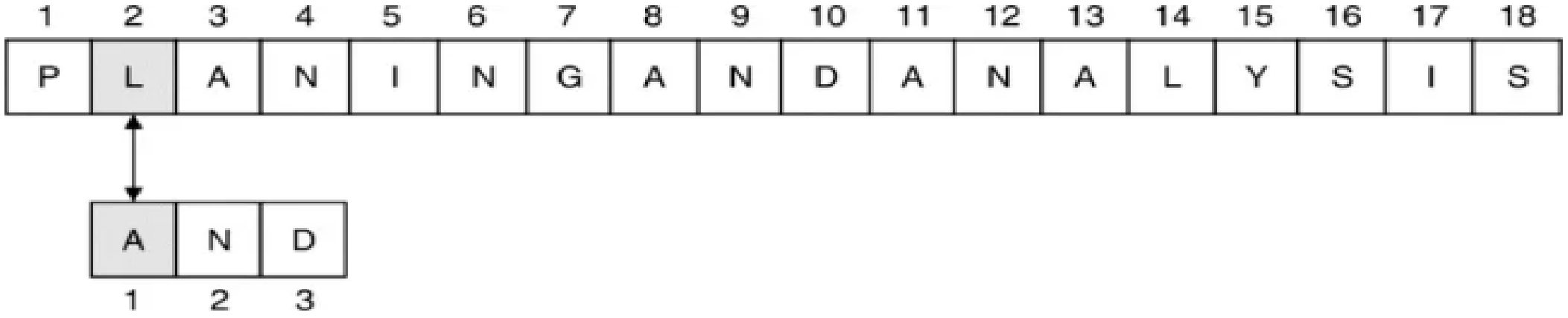
Naive String Matching Algorithm

- This is simple and efficient brute force approach. It compares the first character of pattern with searchable text. If a match is found, pointers in both strings are advanced. If a match is not found, the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text.
- The naïve approach does not require any pre-processing. Given text T and pattern P, it directly starts comparing both strings character by character.
- After each comparison, it shifts pattern string *one position* to the right.
- Following example illustrates the working of naïve string matching algorithm. Here,
  T = PLANINGANDANALYASIS and P = AND
- Here, t_i and p_j are indices of text and pattern respectively.

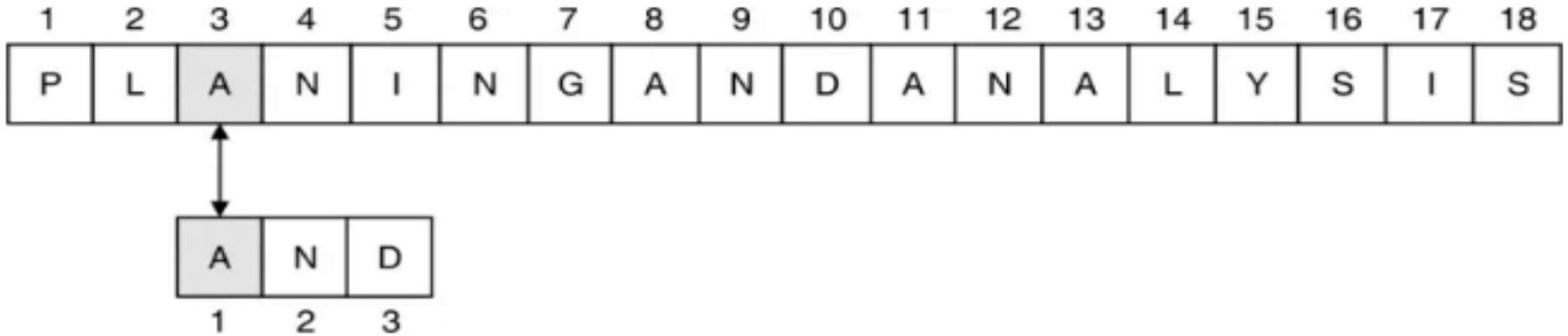
Step 1: T[1] ≠ P[1], so advance text pointer, i.e. t_i++.



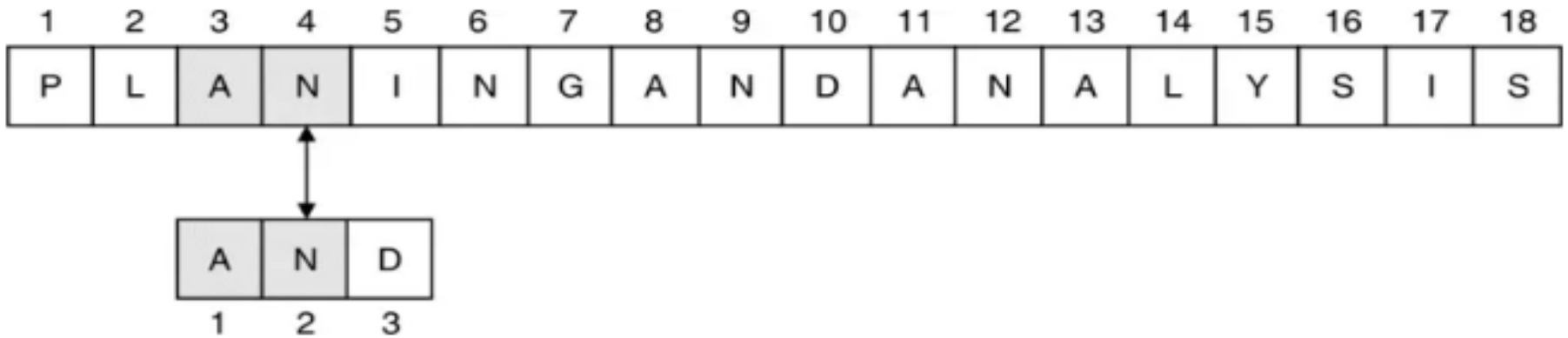
Step 2 : T[2] ≠ P[1], so advance text pointers i.e. t_i++



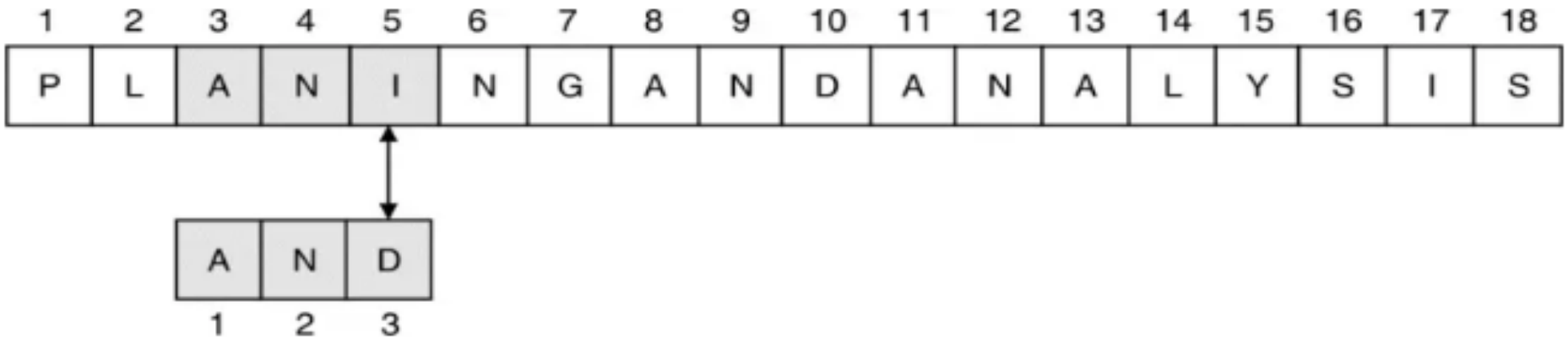
Step 3 : T[3] = P[1], so advance both pointers i.e. t_i++, p_j++



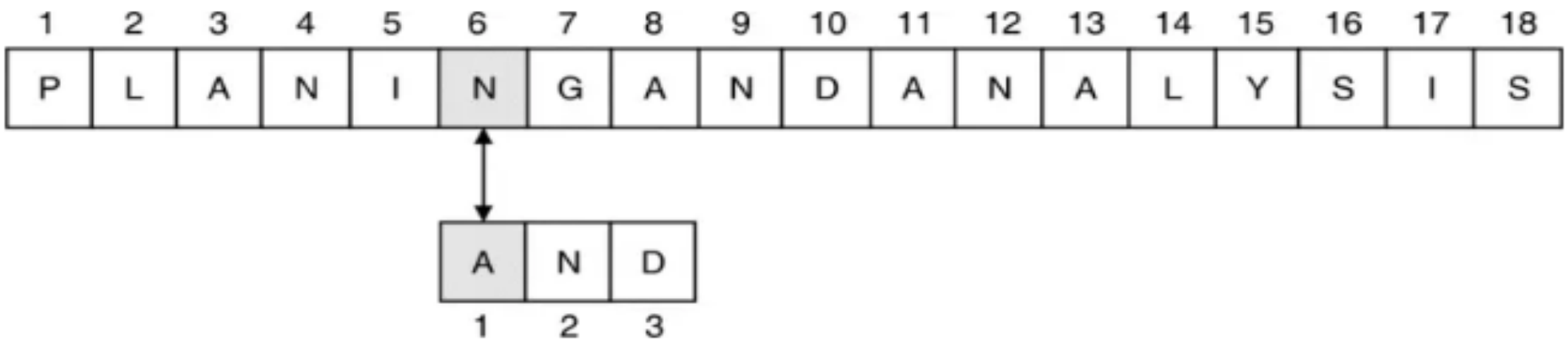
Step 4 : T[4] = P[2], so advance both pointers, i.e. t_i++, p_j++



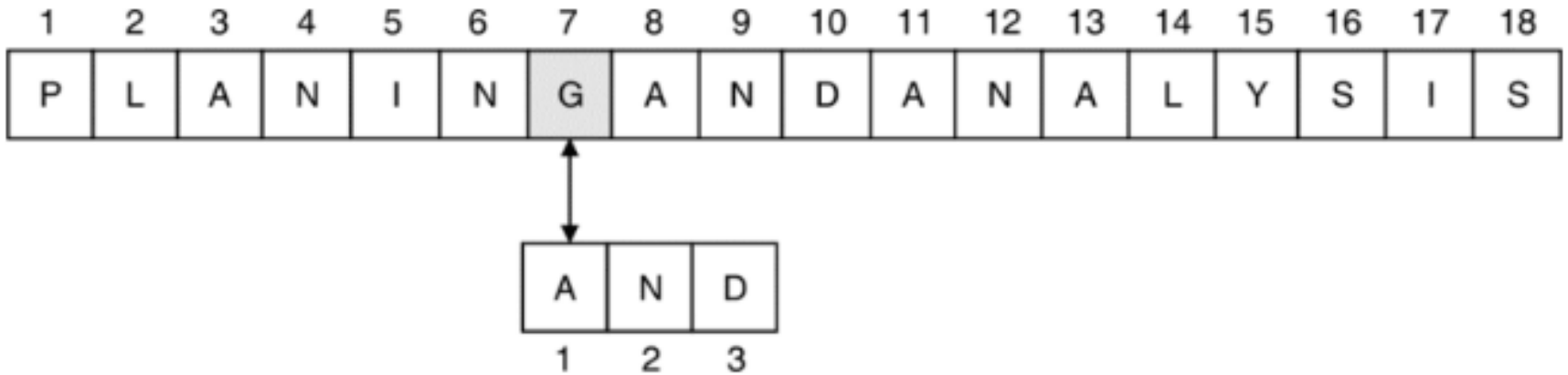
Step 5 : T[5] ≠ P[3], so advance text pointer and reset pattern pointer, i.e. t_i++, p_j = 1



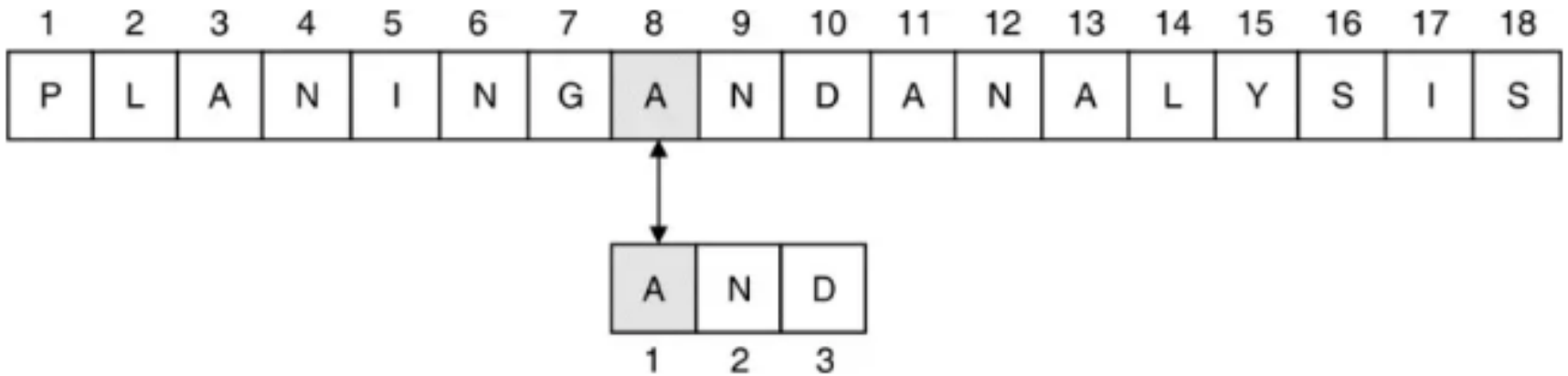
Step 6 : T[6] ≠ P[1], so advance text pointer, i.e. t_i++



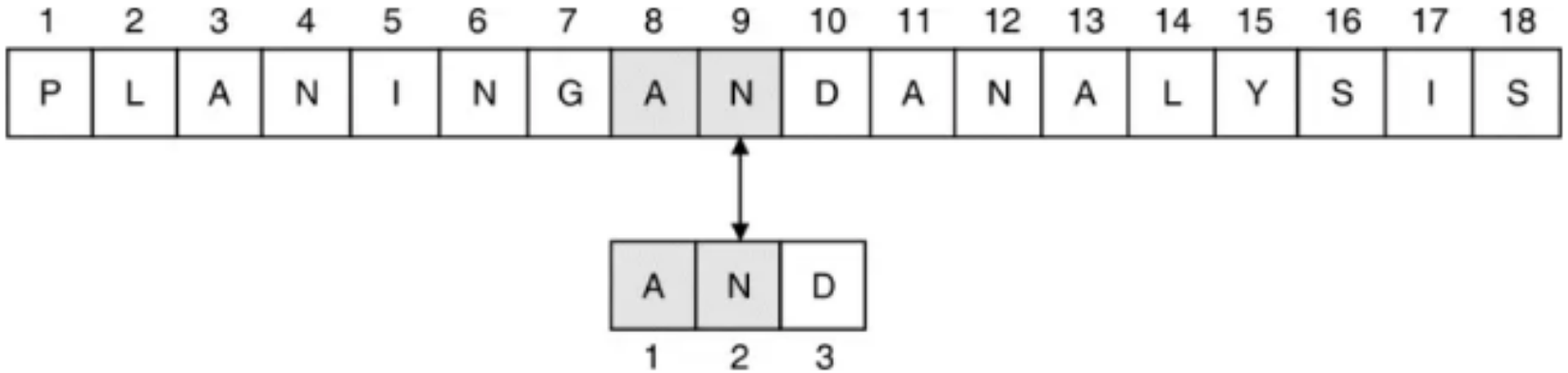
Step 7: T[7] ≠ P[1], so advance text pointer i.e. t_i++



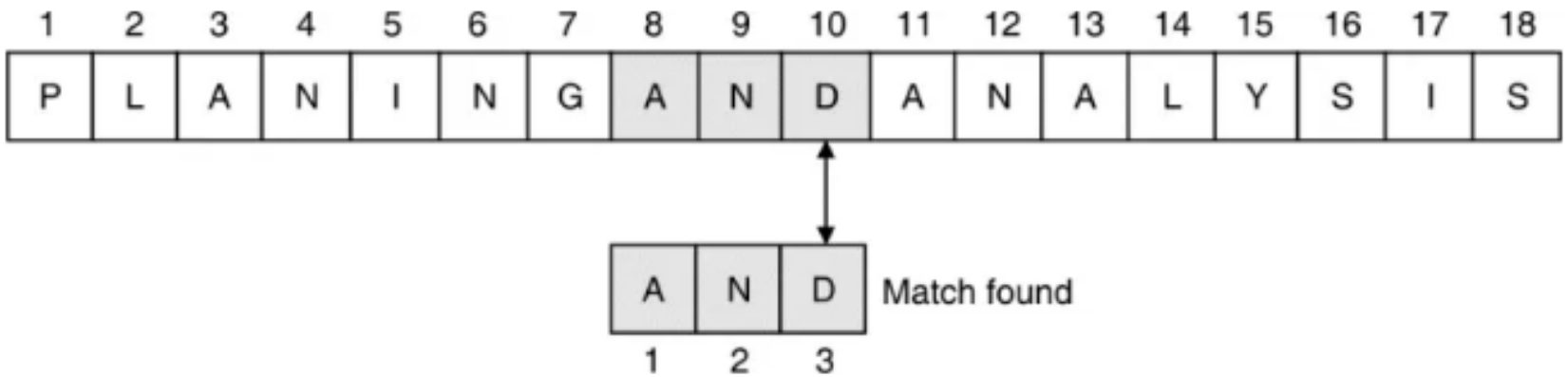
Step 8 : T[8] = P[1], so advance both pointers, i.e. t_i++, p_j++



Step 9 : T[9] = P[2], so advance both pointers, i.e. t_i++, p_j++



Step 10 : T[10] = P[3], so advance both pointers, i.e. t_i++, p_j++



This process continues till the possible comparison in the text string.

Algorithm

- Algorithm NAÏVE_STRING_MATCHING(T, P)
- // T is the text string of length n
- // P is the pattern of length m

- for i ← 0 to n – m do
- if P[1… m] == T[i+1…i+m] them
- print “Match Found”
- end
- End

- Complexity = O(n*m)

Complexity Analysis

(i) Pattern found

- The worst case occurs when the pattern is at last position and there are spurious hits all the way. Example, T = AAAAAAAAAAB, P = AAAB.
- To move pattern one position right, m comparisons are made. Searchable text in T has a length (n – m). Hence, in worst case algorithm runs in O(m*(n – m)) time.

(ii) Pattern not found

- In the best case, the searchable text does not contain any of the prefixes of the pattern. Only one comparison requires moving pattern one position right.
- The algorithm does O(n – m) comparisons.

KMP (Knuth Morris Pratt)

- The worst-case complexity of the Naive algorithm is O(m(n-m+1)). The time complexity of the KMP algorithm is O(n) in the worst case. KMP (Knuth Morris Pratt) Pattern Searching. The Naive pattern-searching algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character.

- Examples:

- 1) txt[] = “AAAAAAAAAAAAAAAAAB”, pat[] = “AAAAB”

- 2) txt[] = “ABABABCABABABCABABABC”, pat[] = “ABABAC” (not a worst case, but a bad case for Naive)

- Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

- Components of KMP Algorithm:
- 1. The Prefix Function (Π)(**Longest Proper Prefix which is Suffix (LPS)**): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

- 2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

Example: Compute Π for the pattern 'p' below:

P:

a	b	a	b	a	c	a

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Example: Given a string 'T' and pattern 'P' as follows:

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:

a	b	a	b	a	c	a

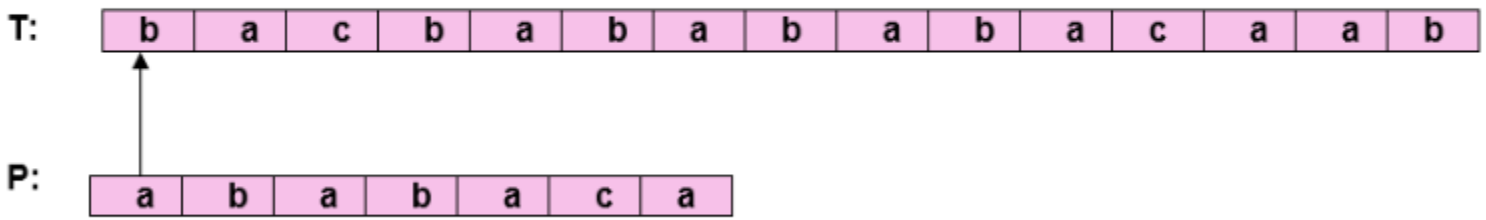
Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Step1: i=1, q=0

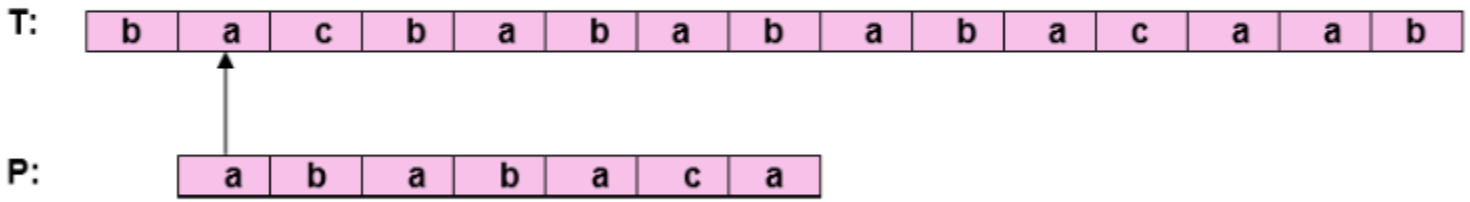
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

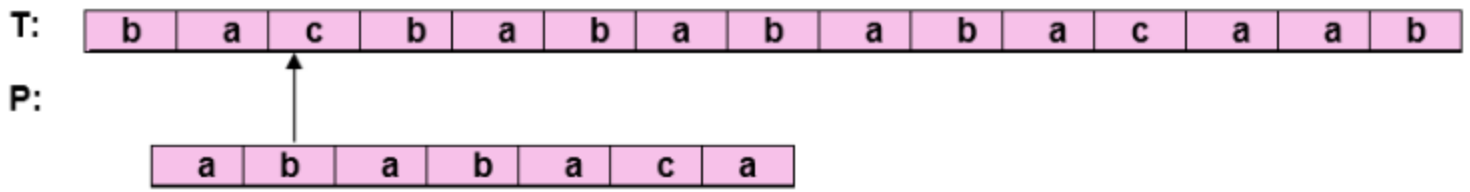
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

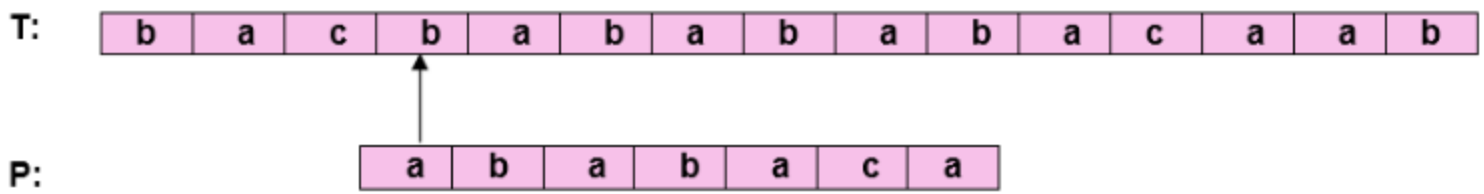
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

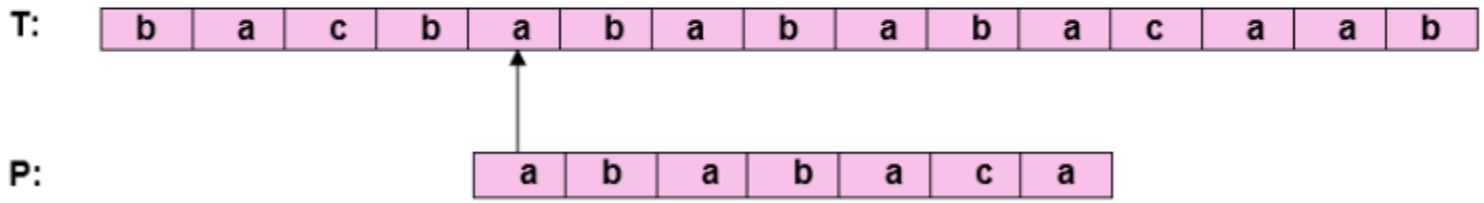
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



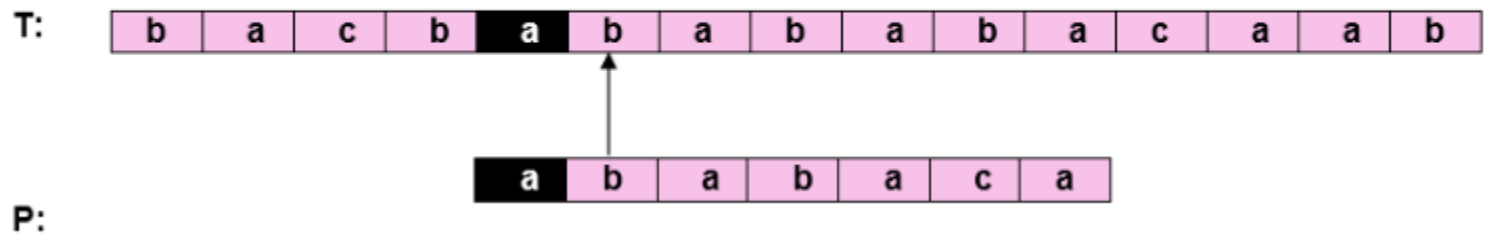
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



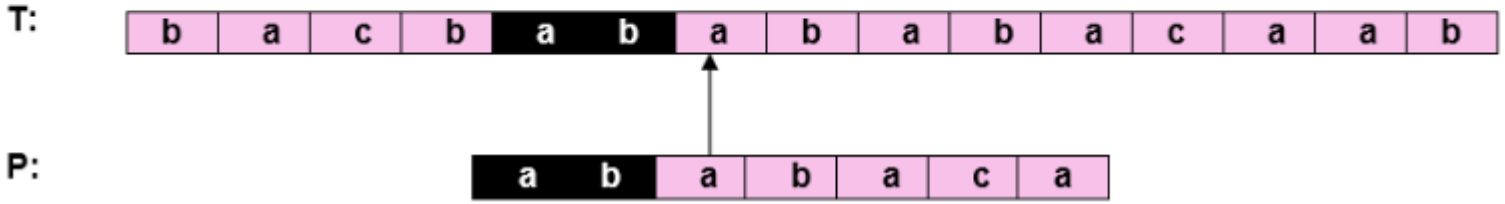
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



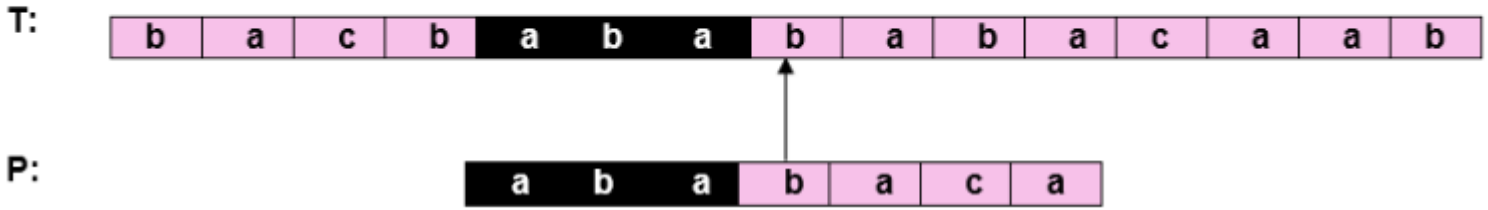
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



Step8: i = 8, q =3

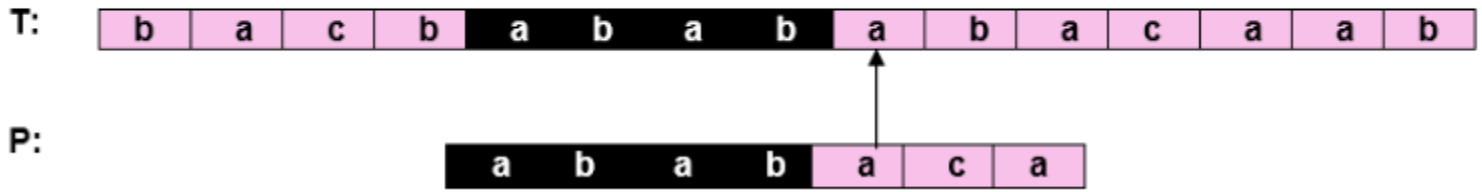
Comparing P [4] with T [8] P [4] matches with T [8]



Step9: i = 9, q = 4

Comparing P [5] with T [9]

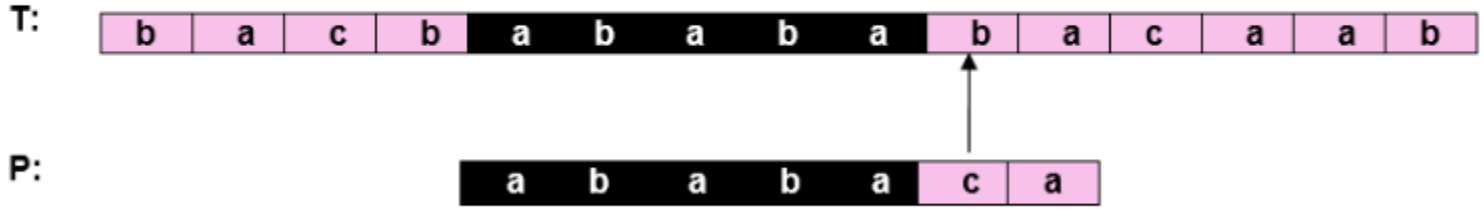
P [5] matches with T [9]



Step10: i = 10, q = 5

Comparing P [6] with T [10]

P [6] doesn't match with T [10]

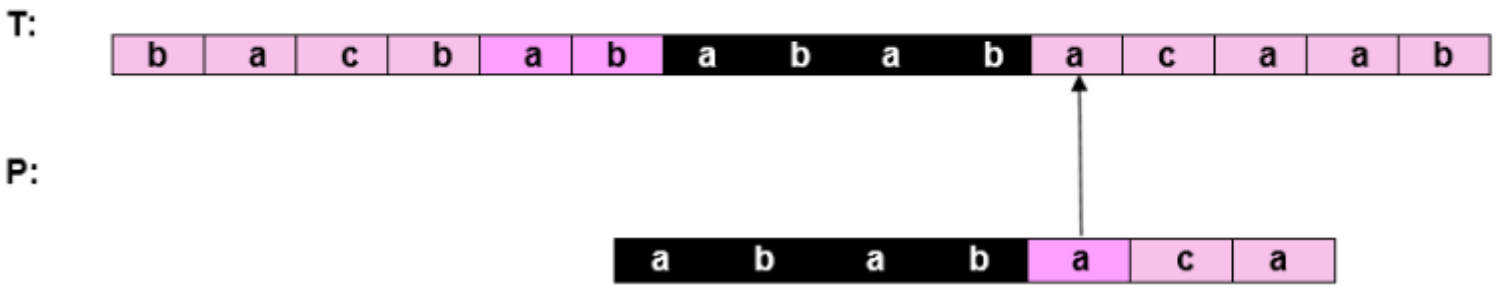


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11]

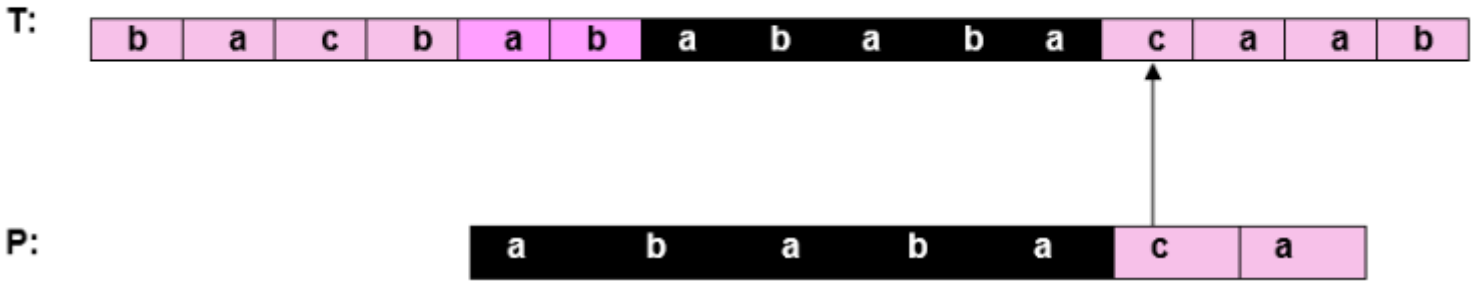
P [5] match with T [11]



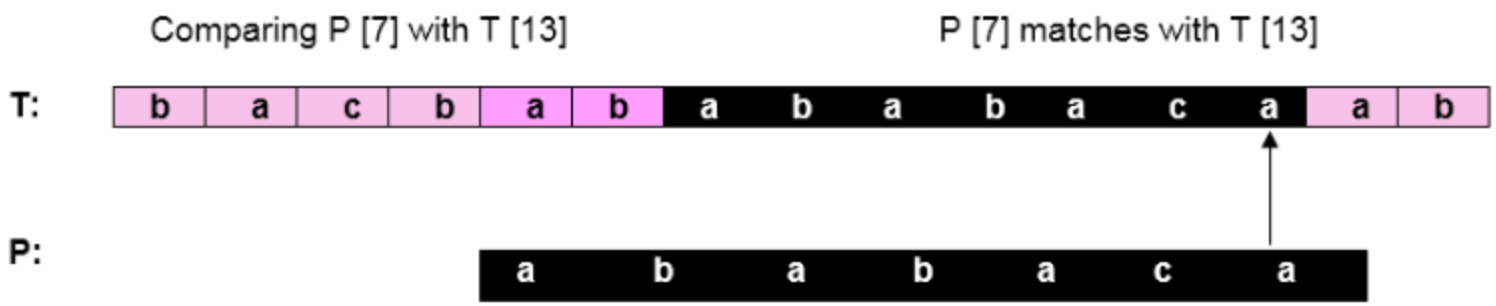
Step12: i = 12, q = 5

Comparing P [6] with T [12]

P [6] matches with T [12]



Step13: i = 3, q = 6



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0                // numbers of characters matched
5. for i ← 1 to n       // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]                // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1              // next character matches
10. If q = m                             // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                          // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Trie (Keyword Tree)

- Strings can essentially be viewed as the most important and common topics for a variety of programming problems. String processing has a variety of real world applications too, such as:
- Search Engines
- Genome Analysis
- Data Analytics
- All the content presented to us in textual form can be visualized as nothing but just strings.
- **Tries:**
- Tries are an extremely special and useful data-structure that are based on the *prefix of a string*. They are used to represent the “Re**trie**val” of data and thus the name Trie.

Prefix : What is prefix:

- The prefix of a string is nothing but any n letters $n \leq |S|$ that can be considered beginning strictly from the starting of a string. For example , the word “abacaba” has the following prefixes:
- a
  ab
  aba
  abac
  abaca
  abacab

- A Trie is a special data structure used to store strings that can be visualized like a graph. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.
- Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

For example , consider the following diagram :

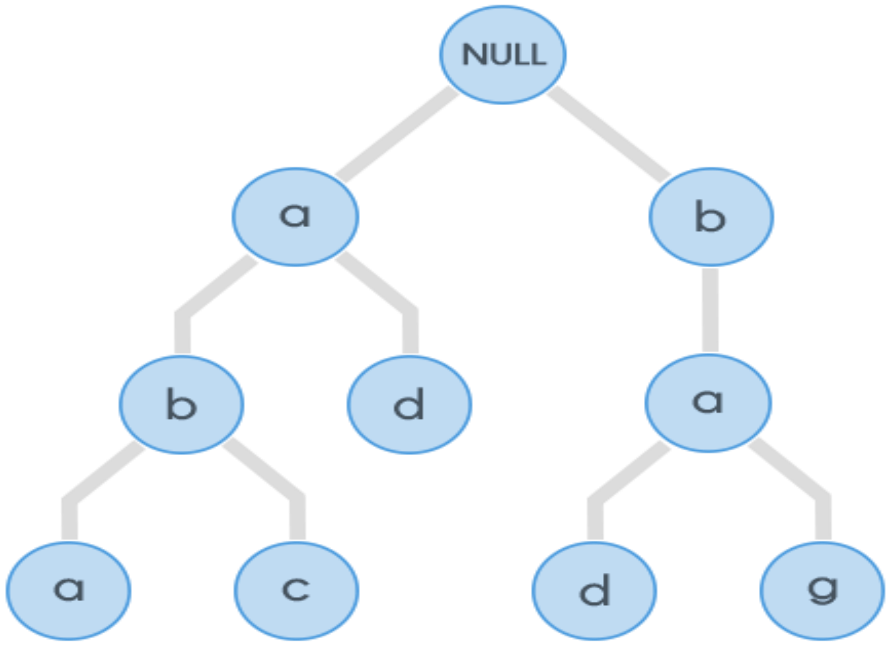
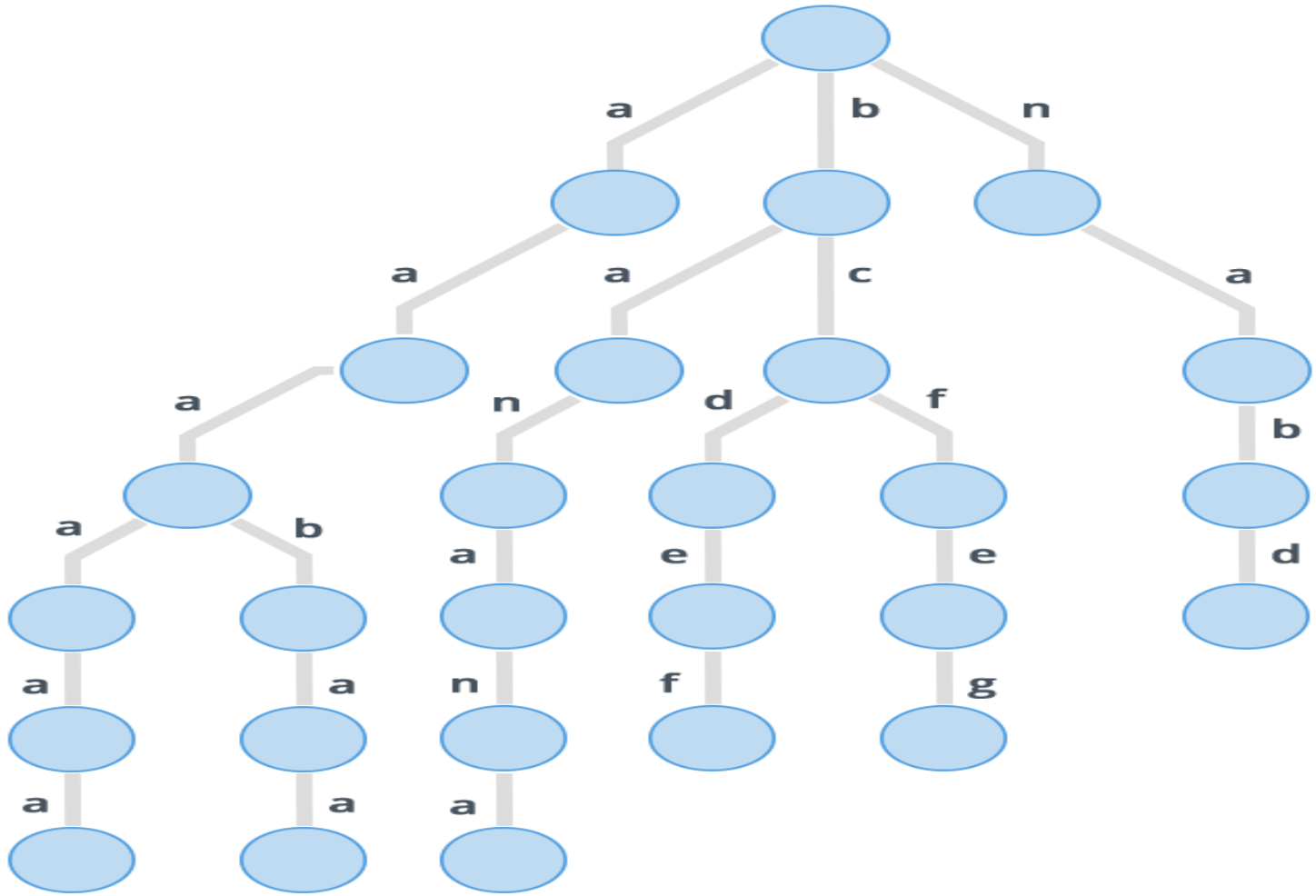


Fig. 1

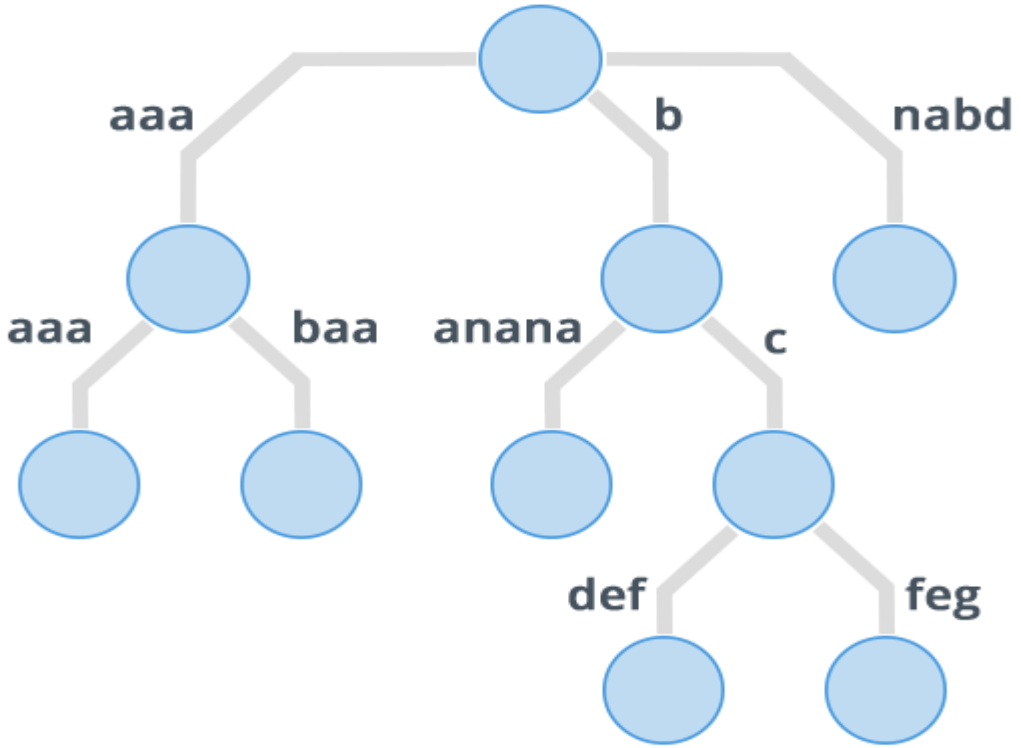
a b
a b a
a b c
a d
b a
b a d
b a g

Suffix Trees

- Suffix tree is a compressed trie of all the suffixes of a given string. Suffix trees help in solving a lot of string related problems like pattern matching, finding distinct substrings in a given string, finding longest palindrome etc.
- Before going to suffix tree, let's first try to understand what a compressed trie is.
  Consider the following set of strings:
  {"banana","nabd","bcdef","bcfeg","aaaaaa","aaabaa" }

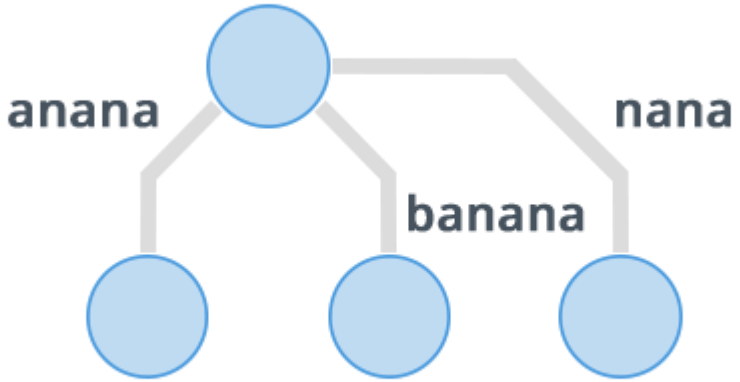

a
b
n
a
a
c
a
a
n
d
f
b
a
b
a
e
e
d
a
a
n
f
g
a
a
a

- a compressed trie for the given set of strings



Implicit Suffix Tree

- In Implicit suffix trees, there are atmost N leaves, while in normal one there should be exactly N leaves. The reason for atmost N leaves is one suffix being prefix of another suffix. Following example will make it clear. Consider the string "banana“



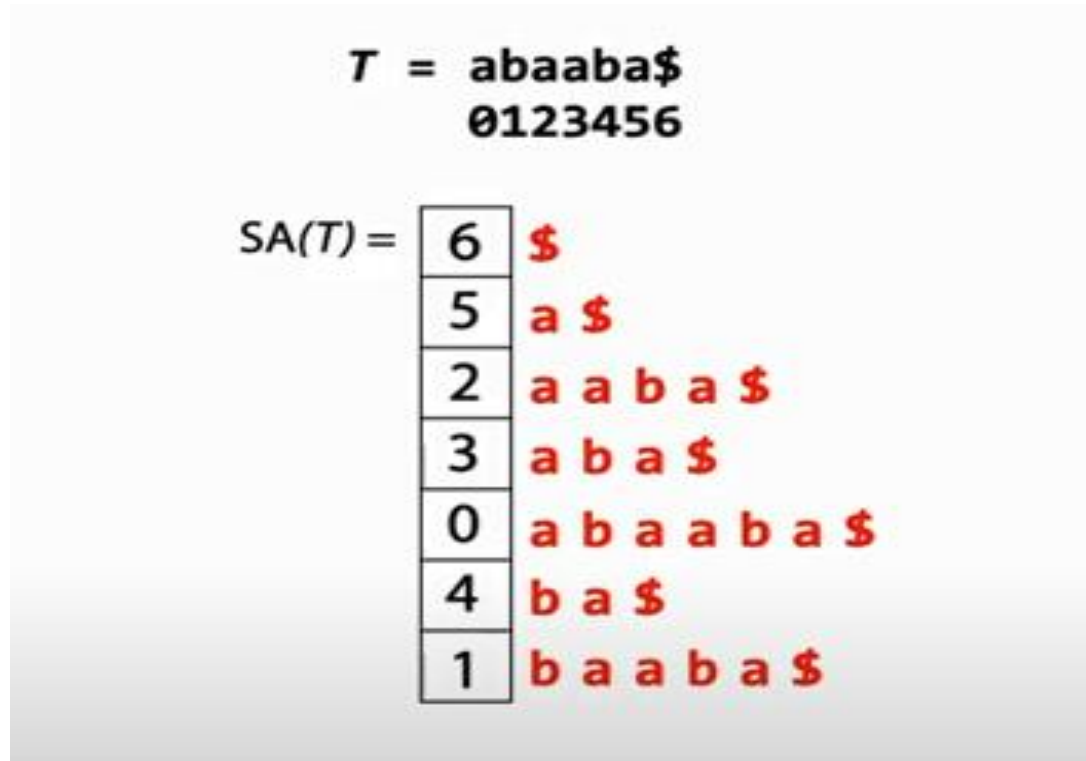
Suffix Arrays

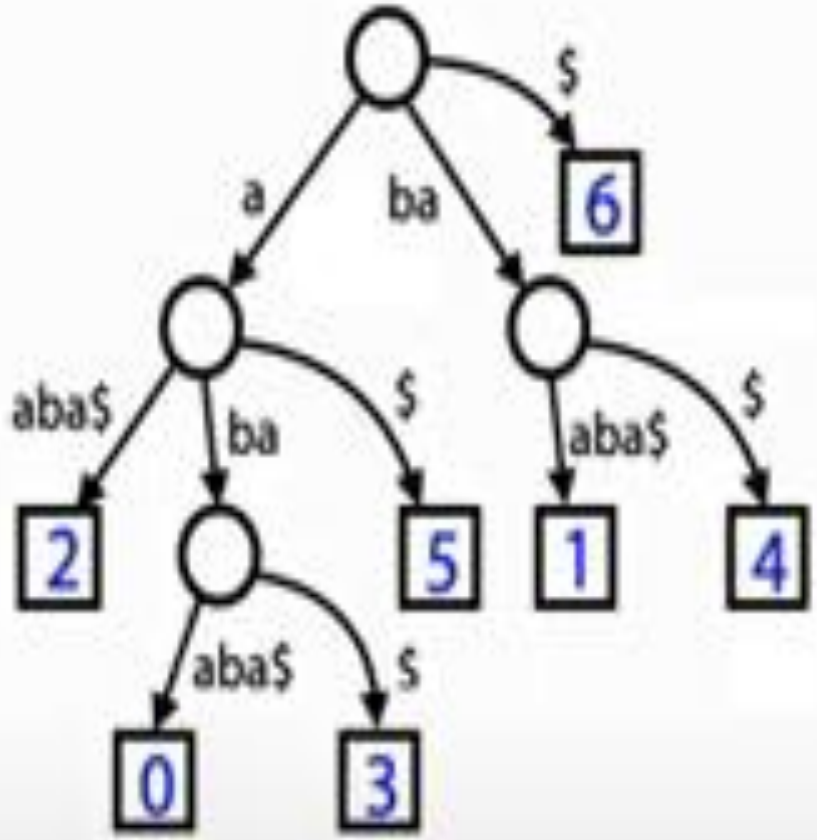
- A Suffix Array is a sorted array of suffixes of a string. Only the indices of suffixes are stored in the string instead of whole strings. For example: Suffix Array of **"banana"** would look like this:
- 5→ a
- 3→ ana
- 1→ anana
- 0→ banana
- 4→ na
- 2→ nana

Suffix array

T = abaaba$
0123456

SA(T) = 

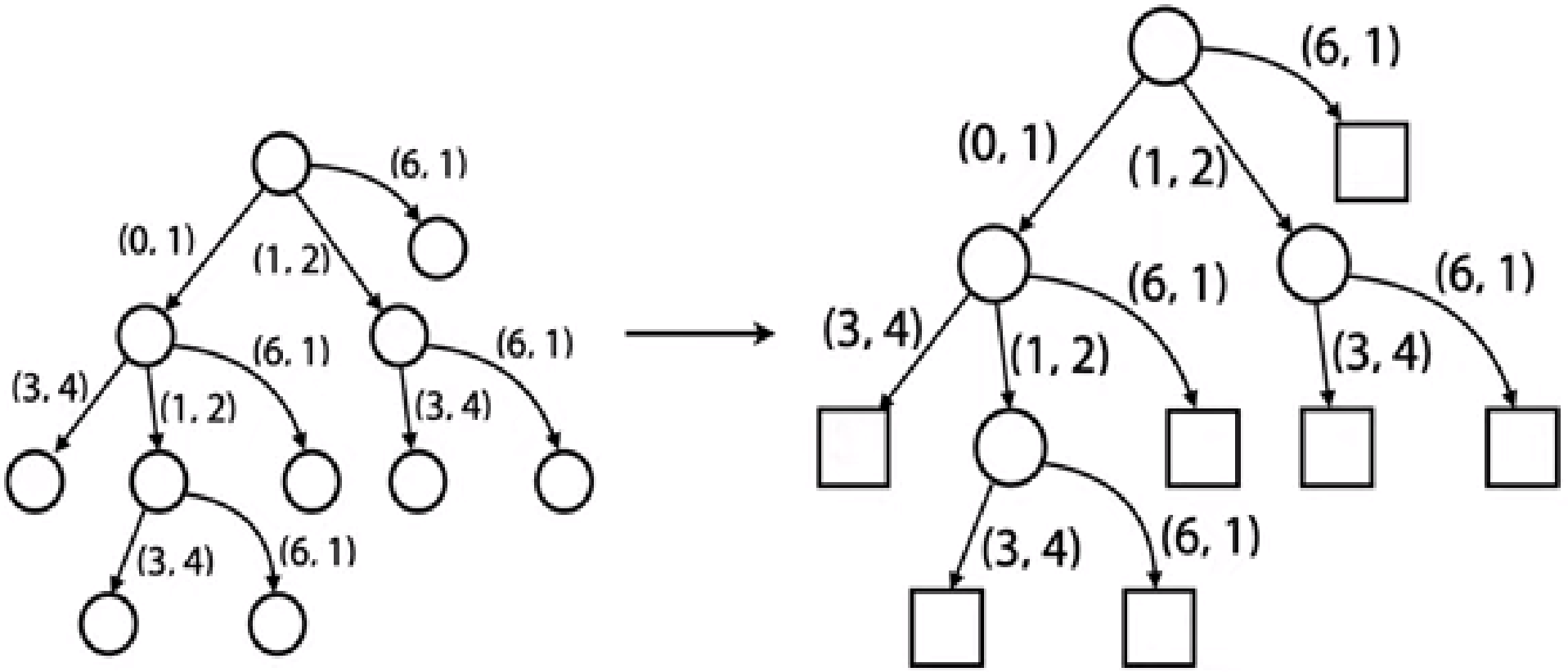

T = abaaba$
0123456
SA(T) =
6 $
5 a $
2 a a b a $
3 a b a $
0 a b a a b a $
4 b a $
1 b a a b a $



T = abaaba$

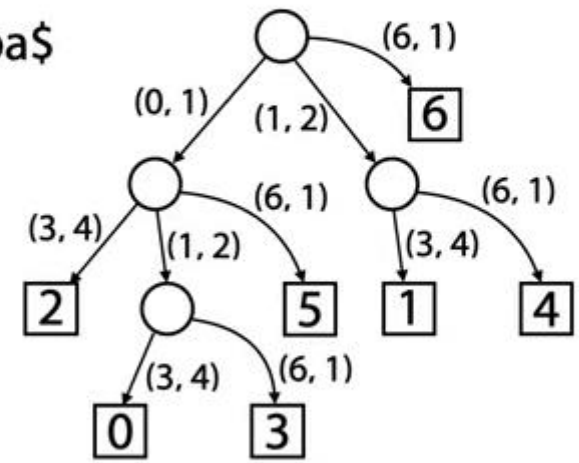
0123456

SA(T) =		
	6	$
	5	a $
	2	a a b a $
	3	a b a $
	0	a b a a b a $
	4	b a $
	1	b a a b a $


(6, 1)
(0, 1)
(1, 2)
(6, 1)
(6, 1)
(3, 4)
(1, 2)
(3, 4)
(3, 4)
(6, 1)
(6, 1)
(0, 1)
(1, 2)
(6, 1)
(6, 1)
(3, 4)
(1, 2)
(3, 4)
(3, 4)
(6, 1)

Suffix tree: leaves hold offsets

T = abaaba$



Computational Geometry :

- Computational geometry is the branch of computer science that studies algorithms for solving geometric problems.

Applications :

- In modern Engineering and Mathematics, computational geometry has applications in following diverse fields:
  - Computer Graphics
  - Robotics
  - VLSI Design
  - Computer Aided Design
  - Molecular Modeling
  - Metallurgy
  - Manufacturing
  - Textile Layouts
  - Forestry
  - Statistics

Computational Geometry :

- The ***input*** to a computational geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order.

- The ***output*** is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

Computational Geometry :
Line Segment Properties :

- **Convex combinations :** A convex combination of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \leq \alpha \leq 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We can also write this as $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Thus p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line.

- **Line segment :** Given two distinct points $p_1$and p_2, the line segment p_1p_2 (over line) is the set of convex combinations of p_1 and p_2. We call p_1 as left endpoint and p_2 as right endpoints of segment p_1p_2.

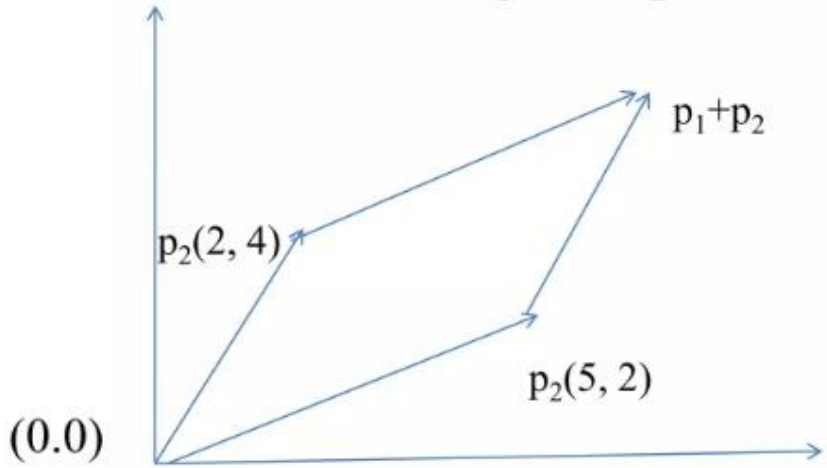
- **Directed segment :** Directed segment p_1p_2 (over right arrow) is referred as the vector p_2 with p_1 as the origin (0, 0).

Computational Geometry :
Line Segment Properties contd…:

- Computational geometry algorithms requires answers to questions about the properties of line segments. These questions are:
    1. Given two **directed segments** p_0p_1 and p_0p_2, is p_0p_1 clockwise from p_0p_2, with respect to their common endpoint p_0?
    2. Given two **line segments** p_0p_1 and p_1p_2, if we traverse p_0p_1,and then p_1p_2, do we make a left turn at point p_1?
    3. Do line segments p_1p_2 and p_3p_4 intersect?

Cross Product :

- Consider vectors p_1 and p_2 as shown below



- We can interpret the **cross product** $p_1 x\ p_2$ as the signed area of the parallelogram formed by points (0,0), p_1, p_2 and $p_1 + p_2$.
- An equivalent but more useful definition gives the cross product as the determinant of a matrix.

Cross Product contd…:

- $p_1 x p_2 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1 = - p_2 x p_1$

- If $\mathbf{p_1 x p_2}$ **is positive** then $\mathbf{p_1}$ **is clockwise from** $\mathbf{p_2}$ with respect to origin (0,0) and if $\mathbf{p_1 x p_2}$ **is negative** then $\mathbf{p_1}$ **is counterclockwise from** $\mathbf{p_2}$ with respect to origin (0,0).

- In above example cross product $p_1 x p_2 = 16$ i.e. positive hence p_1 is clockwise from p_2.

- If cross product is zero then vectors p_1 and p_2 are collinear pointing in either the same or opposite direction.

Determining whether two line segments intersect :

- To determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A line segment p_1p_2 straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. A boundary case arises if p_1 and p_2 lies directly on the line.
- Thus two line segments intersect if and only if either or both of the following conditions hold.
    1. Each segment straddles the line containing other.
    2. An endpoint of one lies on the other segment (Boundary case).
- Following procedures implement this idea:
    1. DIRECTION : computes relative orientations
    2. ON-SEGMENT : determines whether a point known to be collinear with a segment lies on that segment.
    3. SEGMENT-INTERSECT : returns TRUE if segments p_1p_2 and p_3p_4 intersect, otherwise returns FALSE

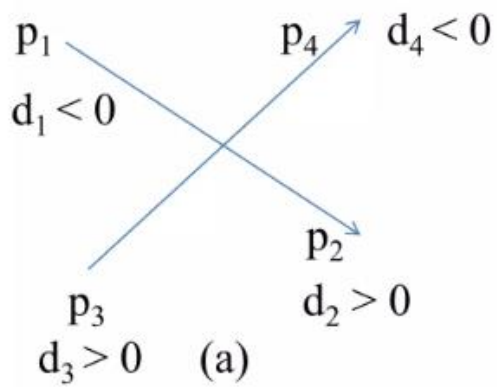
Algorithm DIRECTION (p_i, p_j, p_k)
// p_i and p_j are endpoints of one line segment and p_k is one
// point (endpoint) of the other line segment
{ return $(p_k - p_i) \times (p_j - p_i)$ // cross product O(1)
}

Algorithm ON-SEGMENT (p_i, p_j, p_k)
// p_i and p_j are endpoints of one line segment and p_k is one
// point (endpoint) of the other line segment
{ if $(\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j))$ AND // O(1)
$(\min(y_i, y_j) \leq y_k \leq \max(y_j, y_i))$
return TRUE
else return FALSE
}

Algorithm SEGMENT-INTERSECT (p_1, p_2, p_3 , p_4)
// algorithm returns TRUE if p_1p_2 and p_3p_4 intersect,
// otherwise it returns FALSE

```
{  d1= DIRECTION (p3, p4, p1) // if +ve, p1 is left to p3p4          //O(1)
   d2= DIRECTION (p3, p4, p2) // if - ve, p2 is right to p3p4        //O(1)
   d3= DIRECTION (p1, p2, p3) // if +ve, p3 is left to p1p2          //O(1)
   d4= DIRECTION (p1, p2, p4) // if - ve, p4 is right to p1p2        //O(1)
   if    ((d1 > 0 AND d2 < 0) OR (d1 < 0 AND d2 > 0)) AND           //O(1)
         ((d3 > 0 AND d4 < 0) OR (d3 < 0 AND d4 > 0))
         return   TRUE
   elseif        d1 = 0 AND ON-SEGMENT(p3, p4, p1) return  TRUE; //O(1)
   elseif        d1 = 0 AND ON-SEGMENT(p3, p4, p2) return  TRUE; //O(1)
   elseif        d1 = 0 AND ON-SEGMENT(p1, p2, p3) return  TRUE; //O(1)
   elseif        d1 = 0 AND ON-SEGMENT(p1, p2, p4) return  TRUE; //O(1)
   else  return FALSE;
}  // Algorithm answers Qustion No 3 and its Time Complexity  is O(1)
```

Determining whether two line segments intersect :



$d_1 = (p_1 - p_3) \times (p_4 - p_3)$,
$d_2 = (p_2 - p_3) \times (p_4 - p_3)$
$d_3 = (p_3 - p_1) \times (p_2 - p_1)$,
$d_4 = (p_4 - p_1) \times (p_2 - p_1)$

Given n line segments, find if any two segments intersect

- Given n line segments (p1, q1), (p2, q2), … (pn, qn), find if the given line segments intersect with each other or not.
- We have discussed the problem to detect if two given line segments intersect or not. In this post, we extend the problem. Here we are given n line segments and we need to find out if any two line segments intersect or not.

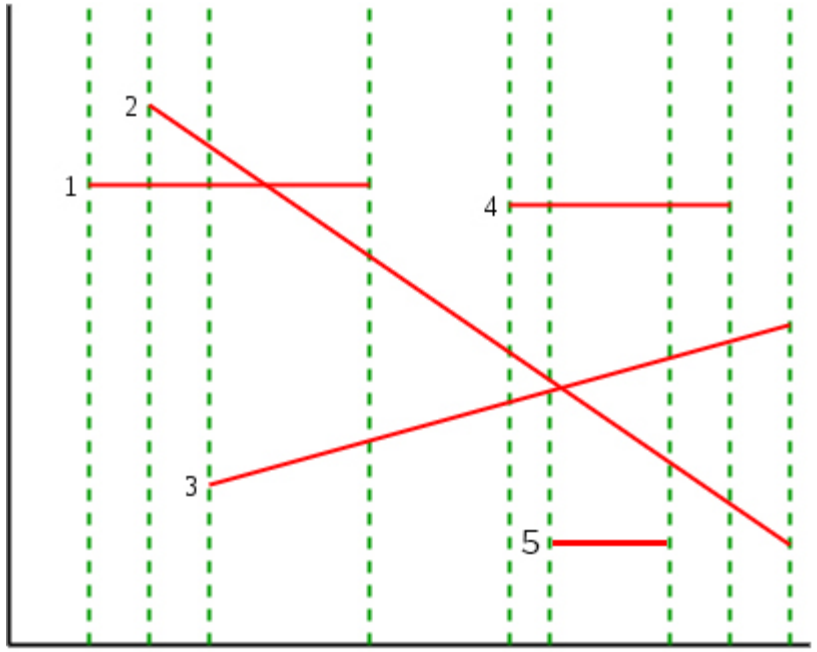
Sweep Line Algorithm

- The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.
  1) Let there be n given lines. There must be 2n end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.
  2) Start from the leftmost point. Do following for every point
  …..**a)** If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to *active* line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.
  ….**b)** If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

- The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm.

- **What data structures should be used for efficient implementation?**
  In step 2, we need to store all active line segments. We need to do following operations efficiently:
  a) Insert a new line segment
  b) Delete a line segment
  c) Find predecessor and successor according to y coordinate values
  The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in O(Logn) time.

- **Example:**
  Let us consider the following example taken from here. There are 5 line segments 1, 2, 3, 4 and 5. The dotted green lines show sweep lines.



Sweep Lines

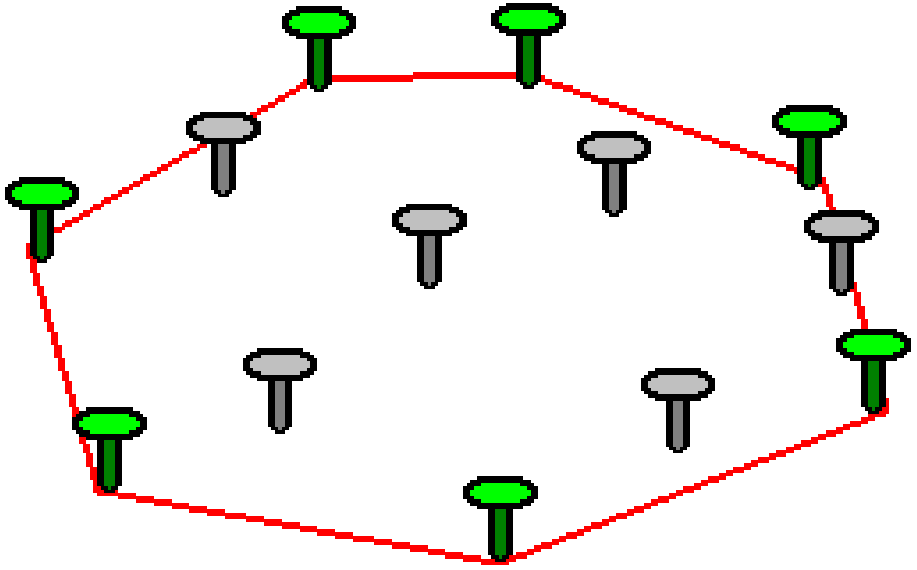
- *Left end point of line segment 1 is processed*: 1 is inserted into the Tree. The tree contains 1. No intersection.
- *Left end point of line segment 2 is processed*: Intersection of 1 and 2 is checked. 2 is inserted into the Tree. Intersection of 1&2 Found ("Note that the above pseudocode returns at this point"). We can continue from here to report all intersection points. The tree contains 1, 2.
- *Left end point of line segment 3 is processed:* Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.
- *Right end point of line segment 1 is processed:* 1 is deleted from the Tree. Intersection of 2 and 3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3.
- *Left end point of line segment 4 is processed*: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.
- *Left end point of line segment 5 is processed*: Intersection of 5 with 3 is checked. No intersection. 5 is inserted into the Tree. The tree contains 2, 4, 3, 5.
- *Right end point of line segment 5 is processed:* 5 is deleted from the Tree. The tree contains 2, 4, 3.
- *Right end point of line segment 4 is processed:* 4 is deleted from the Tree. The tree contains 2, 4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported (Note that the intersection of 2 and 3 is reported again. We can add some logic to check for duplicates). The tree contains 2, 3.
- *Right end point of line segment 2 and 3 are processed:* Both are deleted from tree and tree becomes empty.

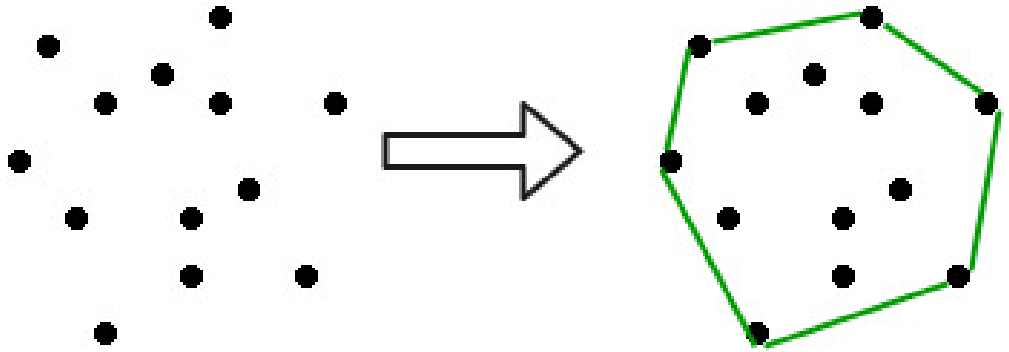
- **Time Complexity:** The first step is sorting which takes O(nLogn) time. The second step process 2n points and for processing every point, it takes O(Logn) time. Therefore, overall time complexity is O(nLogn)

Convex Hull

- **Convex Hull** is the line completely enclosing a set of points in a plane so that there are no concavities in the line. More formally, we can describe it as the smallest convex polygon which encloses a set of points such that each point in the set lies within the polygon or on its perimeter.

- We can visualize what the convex hull looks like by a thought experiment. Imagine that the points are nails sticking out of the plane, take an elastic rubber band, stretch it around the nails and let it go. It will snap around the nails and assume a shape that minimizes its length. The area enclosed by the rubber band is called the convex hull of . This leads to an alternative definition of the convex hull of a finite set of points in the plane: it is the unique convex polygon whose vertices are points from and which contains all points



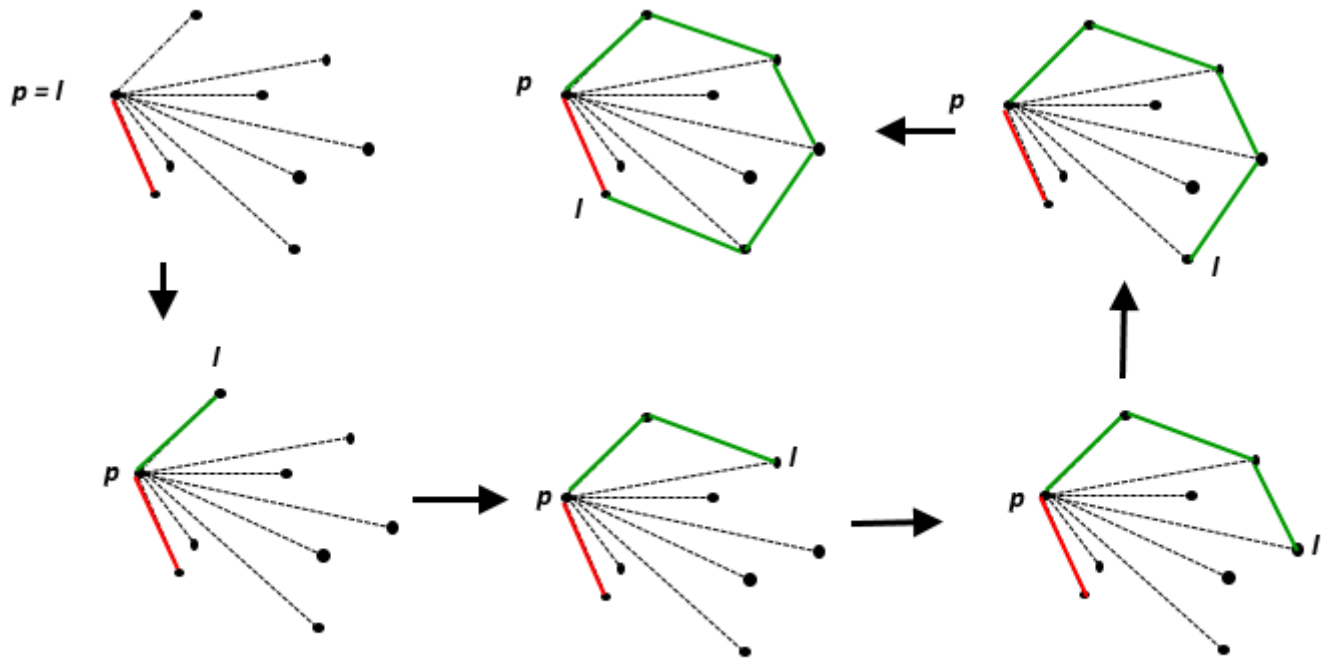


Applications

- **Collision avoidance**: If the convex hull of a car avoids collision with obstacles then so does the car. Since the computation of paths that avoid collision is much easier with a convex car, then it is often used to plan paths.
- **Smallest box**: The smallest area rectangle that encloses a polygon has at least one side flush with the convex hull of the polygon, and so the hull is computed at the first step of minimum rectangle algorithms. Similarly, finding the smallest three-dimensional box surrounding an object depends on the 3D-convex hull.
- **Shape analysis**: Shapes may be classified for the purposes of matching by their "convex deficiency trees", structures that depend for their computation on convex hulls.
- Other practical applications are **pattern recognition, image processing, statistics, geographic information system, game theory, construction of phase diagrams, and static code analysis by abstract interpretation**.

Jarvis March Algorithm (Gift Wrap Algorithm)

- The idea of Jarvis's Algorithm is simple,
- We start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction.
- The big question is, **given a point p as current point, how to find the next point in output?**
- The idea is to use orientation() here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise".



The execution of jarvis's March

Algorithm

- jarvis(S)
- // S is the set of points
- pointOnHull = leftmost point in S //which is guaranteed to be part of the Hull(S)
- i = 0
- repeat
- P[i] = pointOnHull
- endpoint = S[0] // initial endpoint for a candidate edge on the hull
- for j from 1 to |S|
- if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint)
- endpoint = S[j] // found greater left turn, update endpoint
- i = i+1
- pointOnHull = endpoint
- until endpoint == P[0] // wrapped around to first hull point

- Complexity

- Worst case time complexity: Θ(n ^ 2)

Graham Scan Algorithm to find Convex Hull

- **Graham's Scan Algorithm** is an efficient algorithm for finding the convex hull of a finite set of points in the plane with time complexity O(N log N). The algorithm finds all vertices of the convex hull ordered along its boundary. **It uses a stack to detect and remove concavities in the boundary efficiently**.
- It is named after **Ronald Graham**, who published the original algorithm in 1972.
- Worst case time complexity of Jarvis's Algorithm is O(n^2). Using Graham's scan algorithm, we can find Convex Hull in O(nLogn) time.

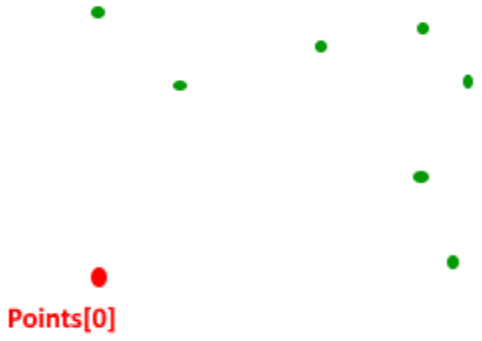
Algorithm

- Let points[0..n-1] be the input array.
- Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottom-most point be P0. Put P0 at first position in output hull.
- Consider the remaining n-1 points and sort them by polor angle in counterclockwise order around points[0]. If polor angle of two points is same, then put the nearest point first.
- After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P0. Let the size of new array be m.
- If m is less than 3, return (Convex Hull not possible)
- Create an empty stack ‘S’ and push points[0], points[1] and points[2] to S.
- Process remaining m-3 points one by one. Do following for every point ‘points[i]’
  Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they don’t make a left turn).
  a) Point next to top in stack
  b) Point at the top of stack
  c) points[i]
  Push points[i] to S
- Print contents of S

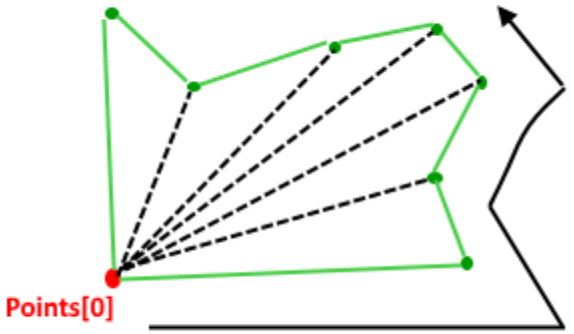
- The above algorithm can be divided in two phases.

- Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points be sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See following diagram).

- What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them

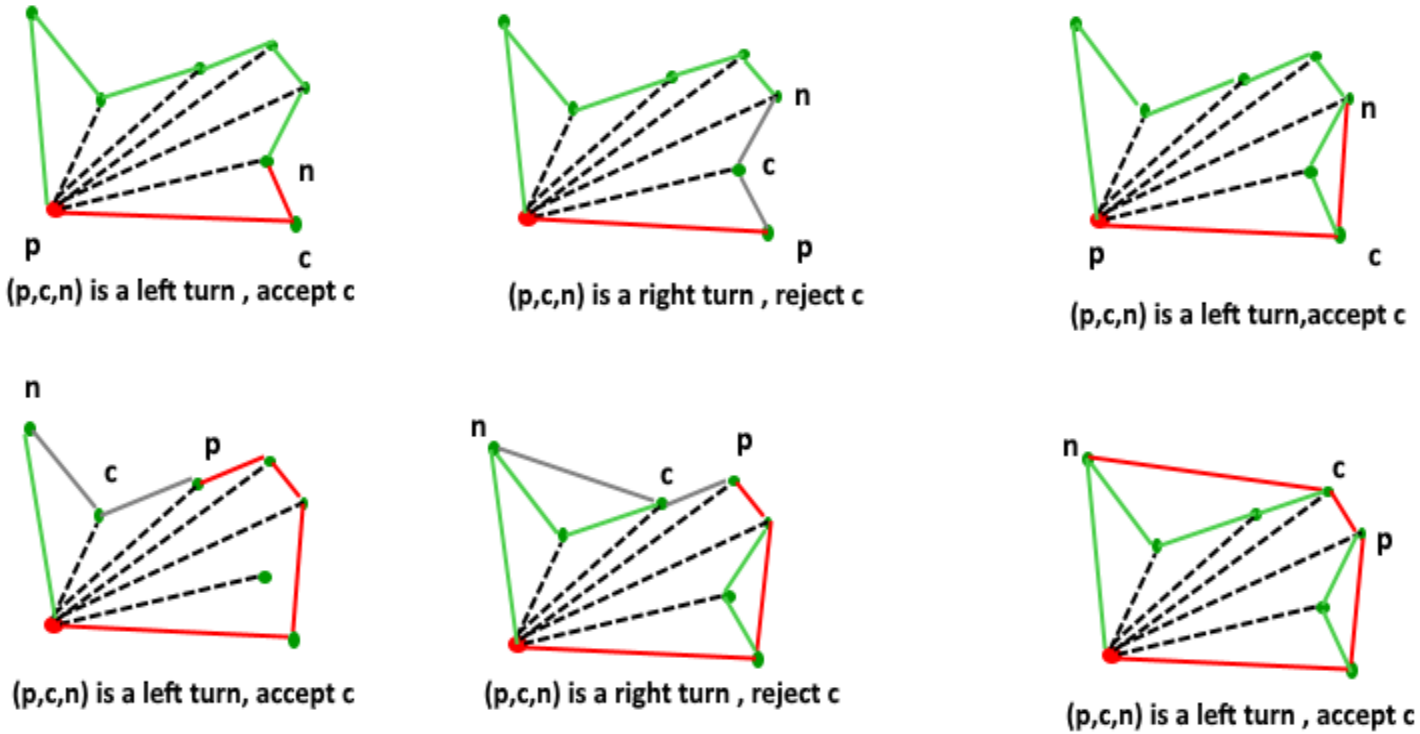


Given points



Points consider according to increasing angle
with respect to points[0] from a simple closed path

- **Phase 2** (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev(p), curr(c) and next(n). If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase



In the above algorithm and below code , a stack of points is used to store convex hull points. With reference to the code ,p is next-to-top in stack, c is top of stack and n is point[i]

- Complexity

- Worst case time complexity: $\Theta(N \log N)$

Approximation Algorithms

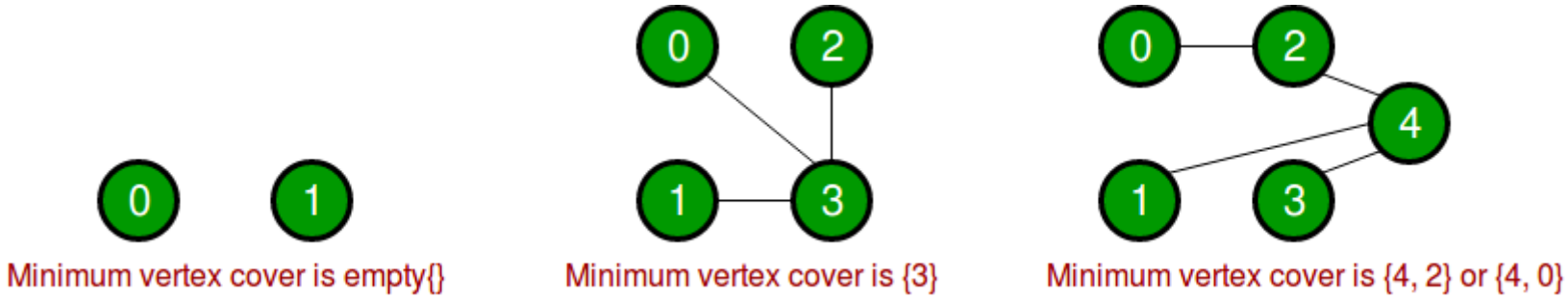
- **Overview :**
  An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time. Such algorithms are called approximation algorithms or heuristic algorithms.

- **Features of Approximation Algorithm :**
  Here, we will discuss the features of the Approximation Algorithm as follows.
- An approximation algorithm guarantees to run in polynomial time though it does not guarantee the most effective solution.
- An approximation algorithm guarantees to seek out high accuracy and top quality solution(say within 1% of optimum)
- Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time

- **Performance Ratios for approximation algorithms :**
- Suppose that we are working on an optimization problem in which each potential solution has a cost, and we wish to find a near-optimal solution. Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost,i.e, the problem can either be a maximization or minimization problem.
- We say that an algorithm for a problem has an appropriate ratio of P(n) if, for any input size n, the cost C of the solution produced by the algorithm is within a factor of P(n) of the cost C* of an optimal solution as follows.
- $\max(C/C^*, C^*/C) <= P(n)$
- For a maximization problem, $0 < C < C^*$, and the ratio of C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate algorithm.
- For a minimization problem, $0 < C^* < C$, and the ratio of C/C^* gives the factor by which the cost of an approximate solution is larger than the cost of an optimal solution.

Approximate Solution for Vertex Cover Problem

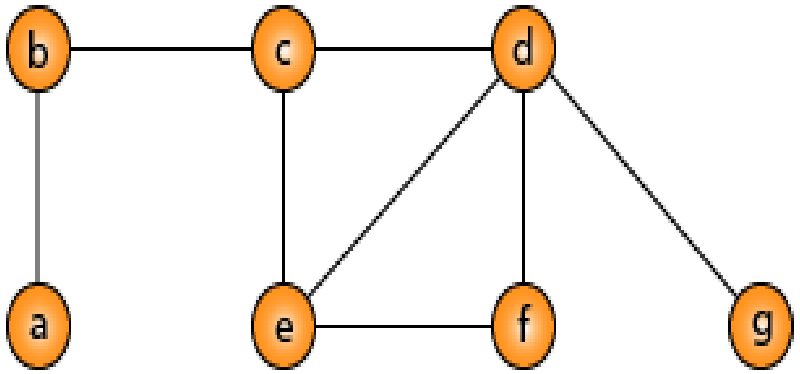
- A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover***.
  The following are some examples.



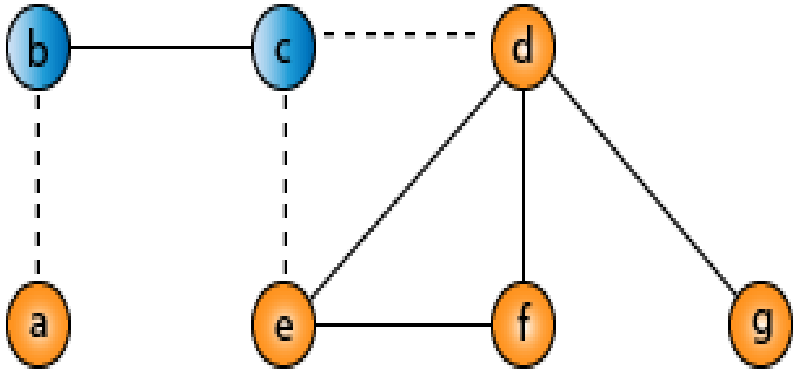
- Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem though.
- Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are:{0,1,2,{0,1},{0,2},{1,2},{0,1,2}} . Using each element of this set check whether these vertices cover all all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

Approximate Algorithm for Vertex Cover

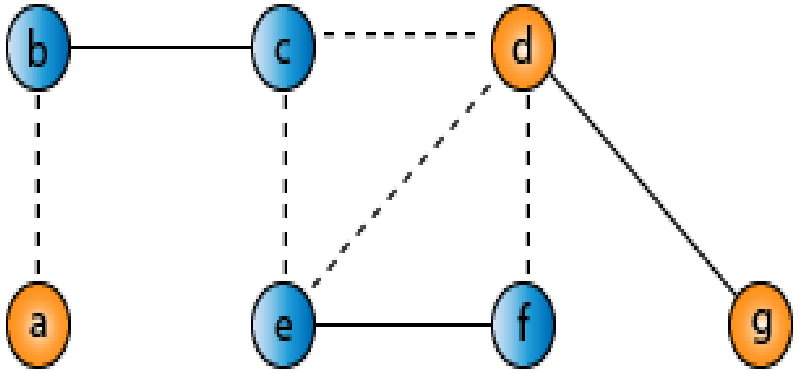
- 1) Initialize the result as {}
- 2) Consider a set of all edges in given graph. Let the set be E.
- 3) Do following while E is not empty
- ...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
- ...b) Remove all edges from E which are either incident on u or v.
- 4) Return result



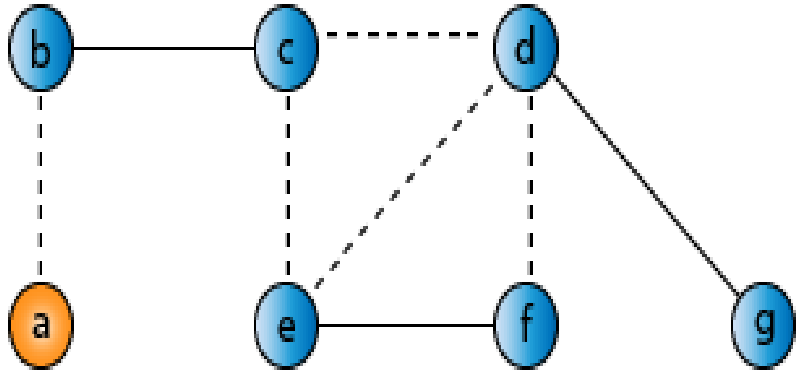
(1)



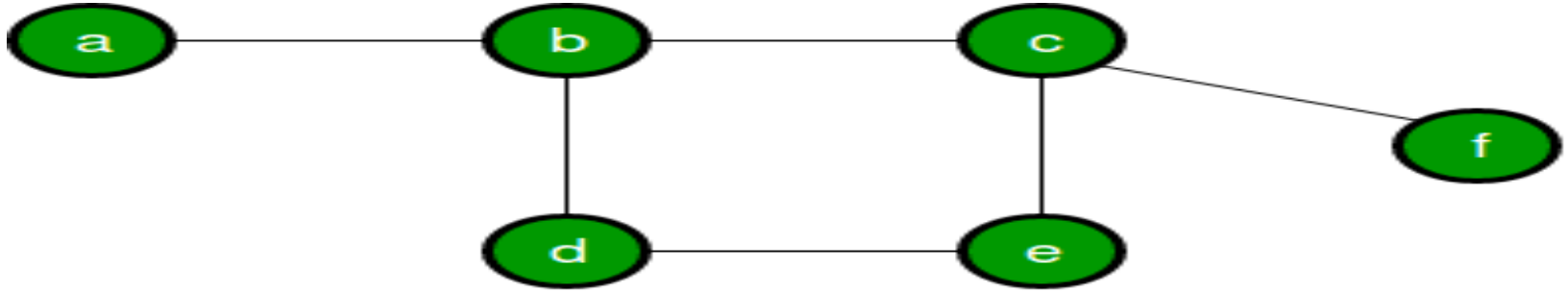
(2)



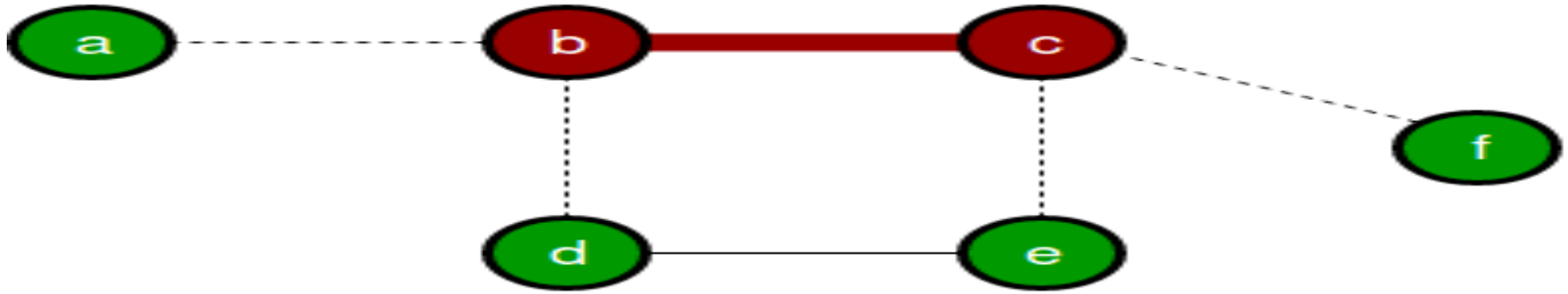
(3)



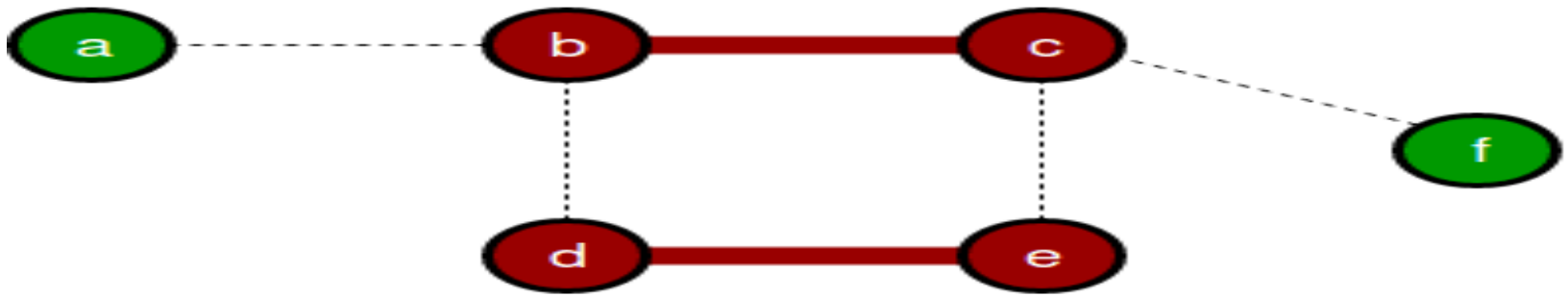
(4)



(a)



(b)



(c)

Minimum Vertex Cover is {b, c, d} or {b, c, e}

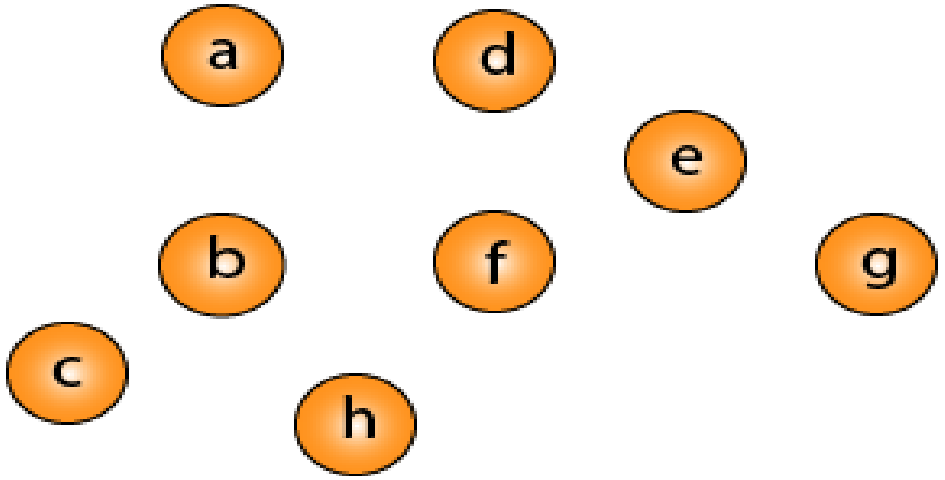
- The time complexity of the above algorithm is **O(V+E)** where V is the number of vertices in the graph and E is the number of edges.

Traveling Salesman Problem (TSP)

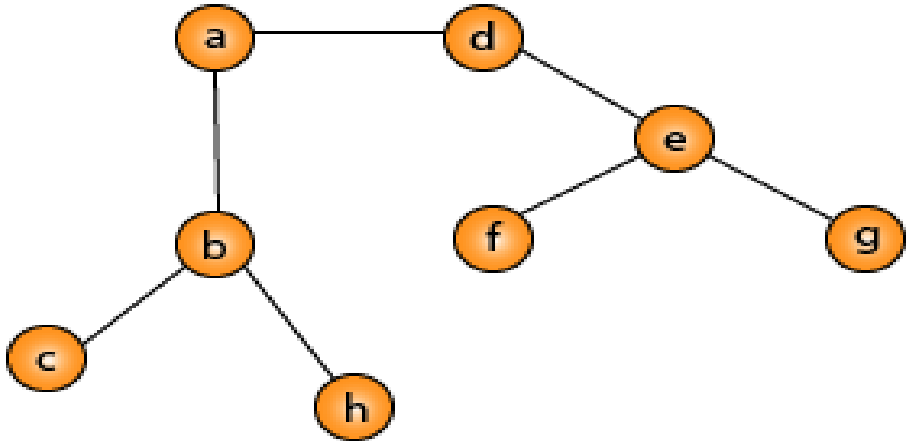
- Travelling Salesman Problem (TSP) : Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
  Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

Approximate TSP algorithm

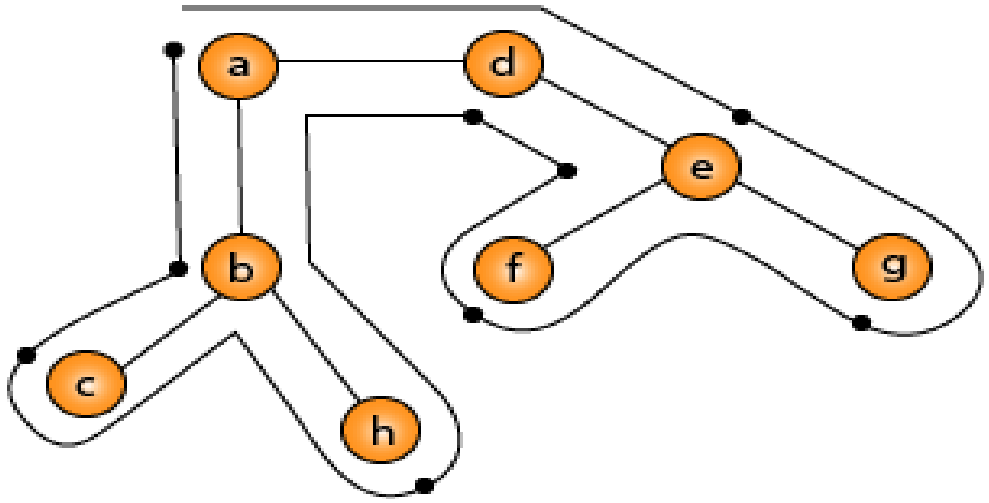
- Approx-TSP (G= (V, E))
- {
- 1. Compute a MST T of G;
- 2. Select any vertex r is the root of the tree;
- 3. Let L be the list of vertices visited in a preorder tree walk of T;
- 4. Return the Hamiltonian cycle H that visits the vertices in the order L;
- }



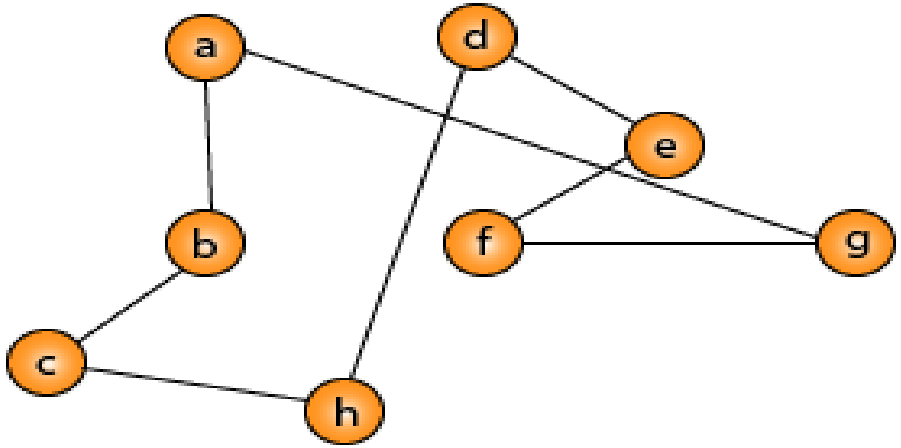
(1) A given set of points



(2) MST T



(3) Full tree walk on T.



(4) A preorder sequence gives a tour H.

Greedy Approximate Algorithm for Set Cover Problem

- Given a universe U of n elements, a collection of subsets of U say S = {S_1, S_2…,S_m} where every subset S_i has an associated cost. Find a minimum cost subcollection of S that covers all elements of U.
- U = {1,2,3,4,5}
- S = {S1,S2,S3}
-
- S1 = {4,1,3}, Cost(S1) = 5
- S2 = {2,5}, Cost(S2) = 10
- S3 = {1,4,3,2}, Cost(S3) = 3

- Output: Minimum cost of set cover is 13 and
- set cover is {S2, S3}

- There are two possible set covers {S1, S2} with cost 15
- and {S2, S3} with cost 13.

- **Why is it useful?**
- Consider General Motors needs to buy a certain amount of varied supplies and there are suppliers that offer various deals for different combinations of materials (Supplier A: 2 tons of steel + 500 tiles for $x; Supplier B: 1 ton of steel + 2000 tiles for $y; etc.). You could use set covering to find the best way to get all the materials while minimizing cost

Approximate Greedy Algorithm

- Let U be the universe of elements, {S1, S2, … Sm} be collection of subsets of U and Cost(S1), C(S2), … Cost(Sm) be costs of subsets.

- 1) Let I represents set of elements included so far. Initialize I = {}

- 2) Do following while I is not same as U.
- a) Find the set Si in {S1, S2, ... Sm} whose cost effectiveness is
- smallest, i.e., the ratio of cost C(Si) and number of newly added
- elements is minimum.
- Basically we pick the set for which following value is minimum.
- Cost(Si) / |Si - I|
- b) Add elements of above picked Si to I, i.e., I = I U Si