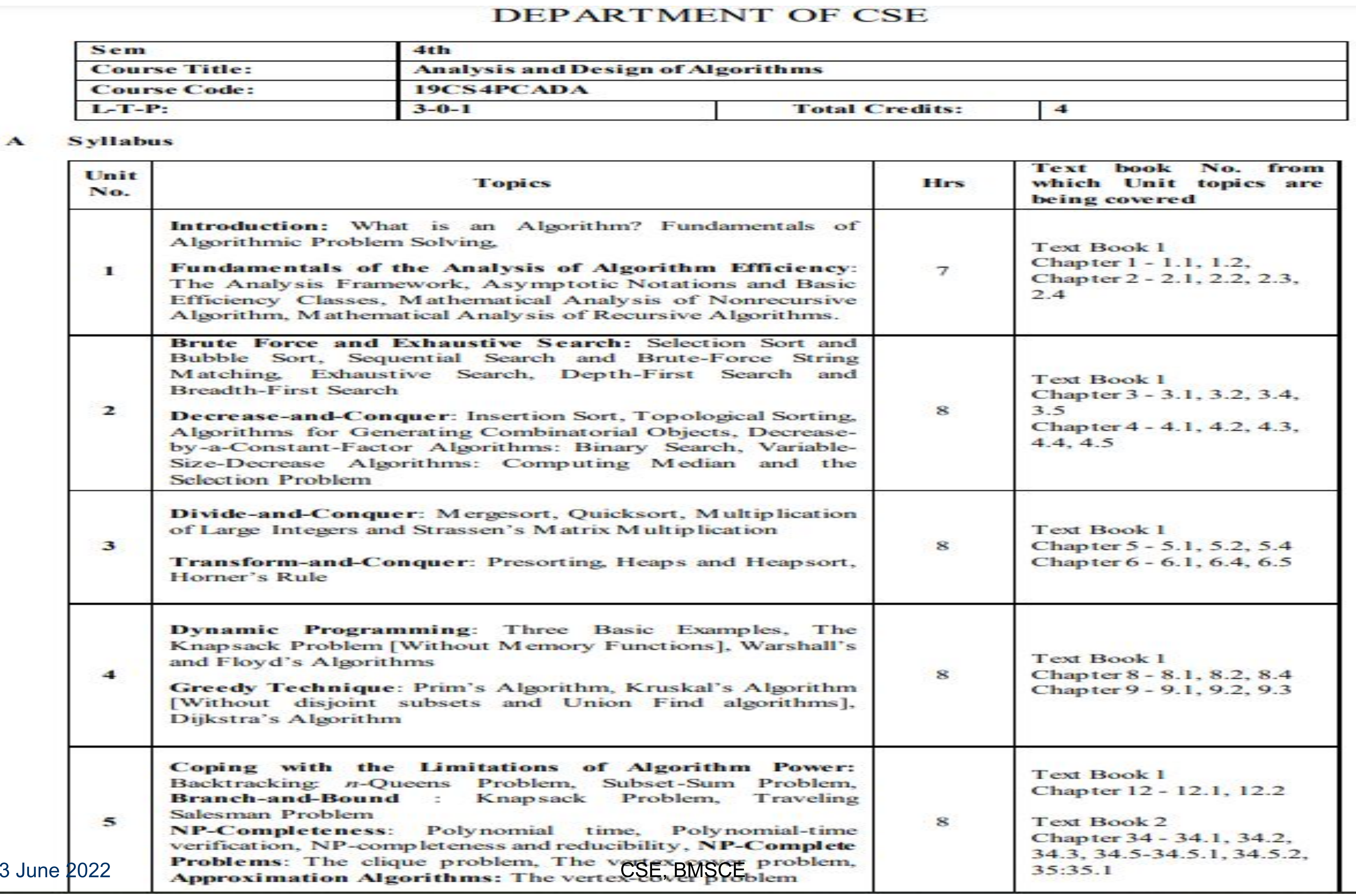
Analysis and Design of Algorithms
(19CS4PCADA)

Google class code: pdhq7op

Dr. Nagarathna N
Professor

Course – Analysis and Design of Algorithms

Unit 2 – Part 1



DEPARTMENT OF CSE

Sem	**4th**		
Course Title:	**Analysis and Design of Algorithms**		
Course Code:	**19CS4PCADA**		
L-T-P:	**3-0-1**	**Total Credits:**	**4**

A Syllabus

Unit No.	Topics	Hrs	Text book No. from which Unit topics are being covered
1	**Introduction:** What is an Algorithm? Fundamentals of Algorithmic Problem Solving. **Fundamentals of the Analysis of Algorithm Efficiency:** The Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Nonrecursive Algorithm, Mathematical Analysis of Recursive Algorithms.	7	Text Book 1 Chapter 1 - 1.1, 1.2, Chapter 2 - 2.1, 2.2, 2.3, 2.4
2	**Brute Force and Exhaustive Search:** Selection Sort and Bubble Sort, Sequential Search and Brute-Force String Matching, Exhaustive Search, Depth-First Search and Breadth-First Search **Decrease-and-Conquer**: Insertion Sort, Topological Sorting, Algorithms for Generating Combinatorial Objects, Decrease-by-a-Constant-Factor Algorithms: Binary Search, Variable-Size-Decrease Algorithms: Computing Median and the Selection Problem	8	Text Book 1 Chapter 3 - 3.1, 3.2, 3.4, 3.5 Chapter 4 - 4.1, 4.2, 4.3, 4.4, 4.5
3	**Divide-and-Conquer**: Mergesort, Quicksort, Multiplication of Large Integers and Strassen's Matrix Multiplication **Transform-and-Conquer**: Presorting, Heaps and Heapsort, Horner's Rule	8	Text Book 1 Chapter 5 - 5.1, 5.2, 5.4 Chapter 6 - 6.1, 6.4, 6.5
4	**Dynamic Programming**: Three Basic Examples, The Knapsack Problem [Without Memory Functions], Warshall's and Floyd's Algorithms **Greedy Technique**: Prim's Algorithm, Kruskal's Algorithm [Without disjoint subsets and Union Find algorithms], Dijkstra's Algorithm	8	Text Book 1 Chapter 8 - 8.1, 8.2, 8.4 Chapter 9 - 9.1, 9.2, 9.3
5	**Coping with the Limitations of Algorithm Power:** Backtracking: *n*-Queens Problem, Subset-Sum Problem, **Branch-and-Bound** : Knapsack Problem, Traveling Salesman Problem **NP-Completeness**: Polynomial time, Polynomial-time verification, NP-completeness and reducibility, **NP-Complete Problems**: The clique problem, The vertex-cover problem, **Approximation Algorithms:** The vertex-cover problem	8	Text Book 1 Chapter 12 - 12.1, 12.2 Text Book 2 Chapter 34 - 34.1, 34.2, 34.3, 34.5-34.5.1, 34.5.2, 35:35.1

Algorithm Design Techniques

Various design techniques exist:

- Classifying algorithms based on design ideas or commonality
- General problem solving strategies
  - Brute force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Dynamic Programming
  - Greedy technique
  - Back tracking

Brute Force Technique

- Brute Force – the simplest of the design strategies
- Is a straight forward approach to **solving a problem**, usually directly **based on the problem's statement** and definition of the concepts involved.
- Just do it – the brute-force strategy is easiest to apply
- Results in an algorithm that can be improved with a modest amount of time

Brute Force Technique

- Selection
- Bubble Sort
- Sequential Search

Brute Force Technique

Selection sort: repeatedly pick the smallest element to append to the result.

Selection sort is to repetitively pick up the smallest element and put it into the correct position:

- Find the smallest element, and put it to the first position.
- Find the next smallest element, and put it to the second position.
- Repeat until all elements are in the correct positions.

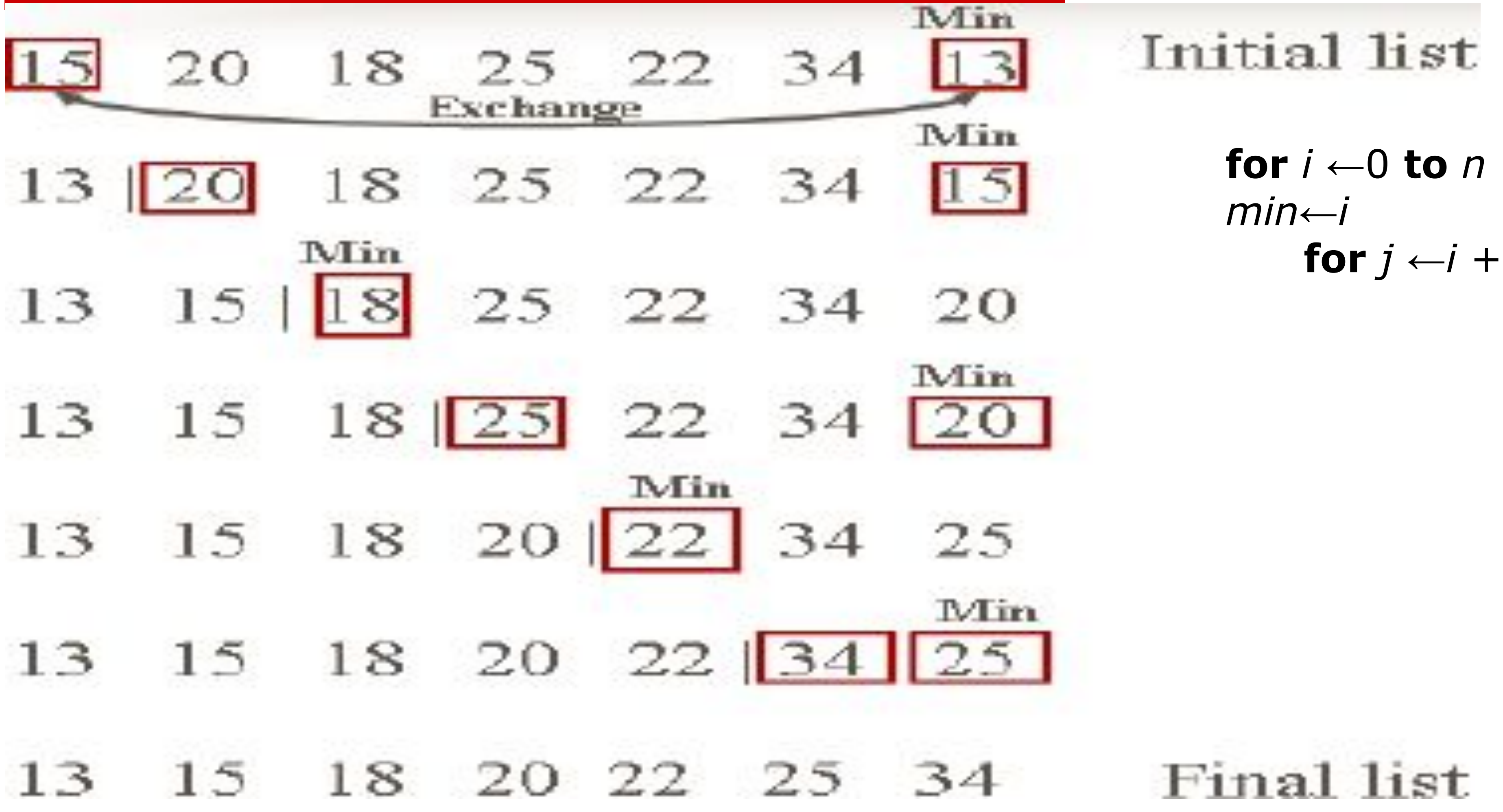
Selection Sort

Scan the list repeatedly to find the elements, one at a time, in an nondecreasing order

- On the i^{th} pass through the list, search for the smallest item among the last (n-i) elements and swap it with A[i]. After (n-1) passes the list is sorted.

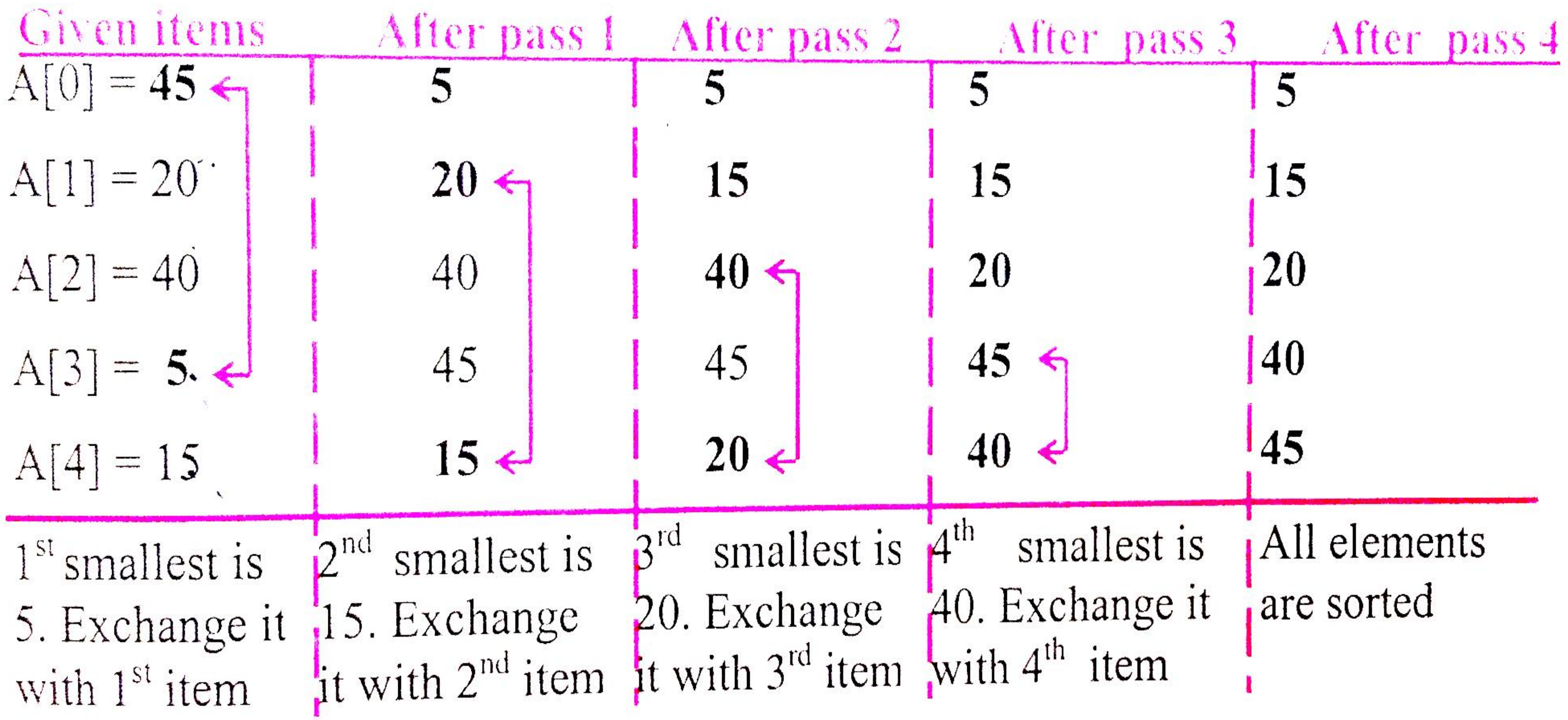
- Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped.

Example: Selection Sort



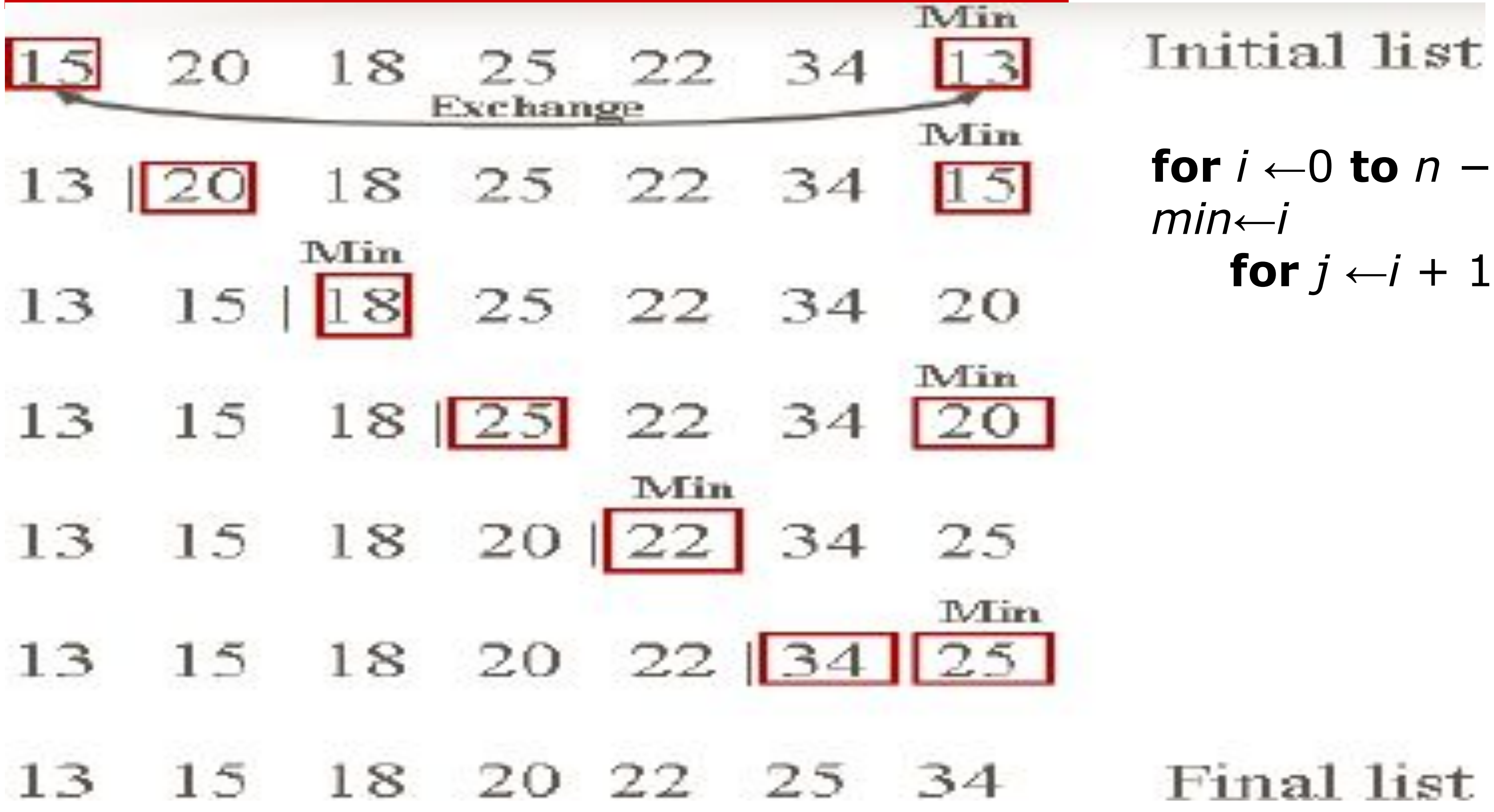
```
for $i \leftarrow 0$ to $n - 2$ {
$min \leftarrow i$
    for $j \leftarrow i + 1$ to $n - 1$
```

Example: Selection Sort



Given items	After pass 1	After pass 2	After pass 3	After pass 4
A[0] = 45	5	5	5	5
A[1] = 20	20	15	15	15
A[2] = 40	40	40	20	20
A[3] = 5	45	45	45	40
A[4] = 15	15	20	40	45
1st smallest is 5. Exchange it with 1st item	2nd smallest is 15. Exchange it with 2nd item	3rd smallest is 20. Exchange it with 3rd item	4th smallest is 40. Exchange it with 4th item	All elements are sorted

Example: Selection Sort



```
for $i \leftarrow 0$ to $n - 2$ {
$min \leftarrow i$
    for $j \leftarrow i + 1$ to $n - 1$
```

Selection Sort Algorithm

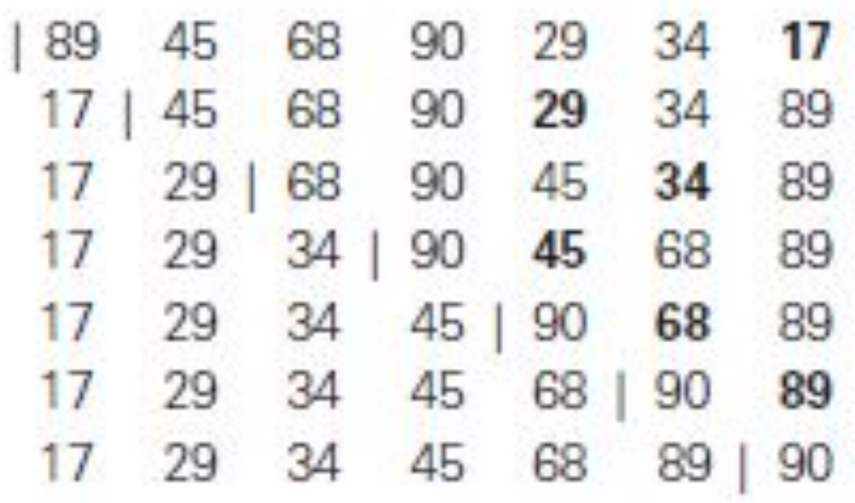
```
ALGORITHM SelectionSort(A[0..n − 1])
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ←0 to n − 2 {
min←i
    for j ←i + 1 to n − 1 {
        if A[j ]<A[min]
            min←j
                    }
swap A[i] and A[min]
}
```

Question

- Sort the elements 89, 45, 68, 90, 29, 34, 17 using selection sort

Answer

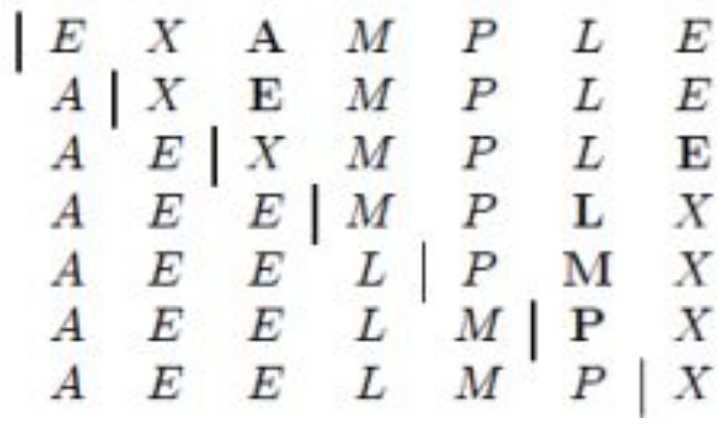
- Sort the elements 89, 45, 68, 90, 29, 34, 17 using selection sort



| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Question

- Sort the list “E X A M P L E” in alphabetical order by selection sort.



```
| E   X   A   M   P   L   E
  A | X   E   M   P   L   E
  A   E | X   M   P   L   E
  A   E   E | M   P   L   X
  A   E   E   L | P   M   X
  A   E   E   L   M | P   X
  A   E   E   L   M   P | X
```

Analysis of Selection Sort

- The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j\ ]<A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}[(n-1)-(i+1)+1]=\sum_{i=0}^{n-2}(n-1-i)=\frac{(n-1)n}{2}.$$

- Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Question

If there are n elements to be sorted using Selection Sort then how many Swap operations will be carried out.

Answer

If there are n elements to be sorted using Selection Sort then how many Swap operations will be carried out.

- Answer: The number of key swaps is only*(n)*, or, more precisely, $n - 1$(one for each repetition of the *i* loop).This property distinguishes selection sort positively from many other sorting algorithms.

Selection Sort

- https://www.geeksforgeeks.org/selection-sort/

Brute Force Technique

Sorting
- Selection Sort
- Bubble Sort

Search
- Sequential Search
- String-Matching problem (or String pattern search)

Brute Force Technique

Bubble sort: repetitively compares adjacent pairs of elements and swaps if necessary.

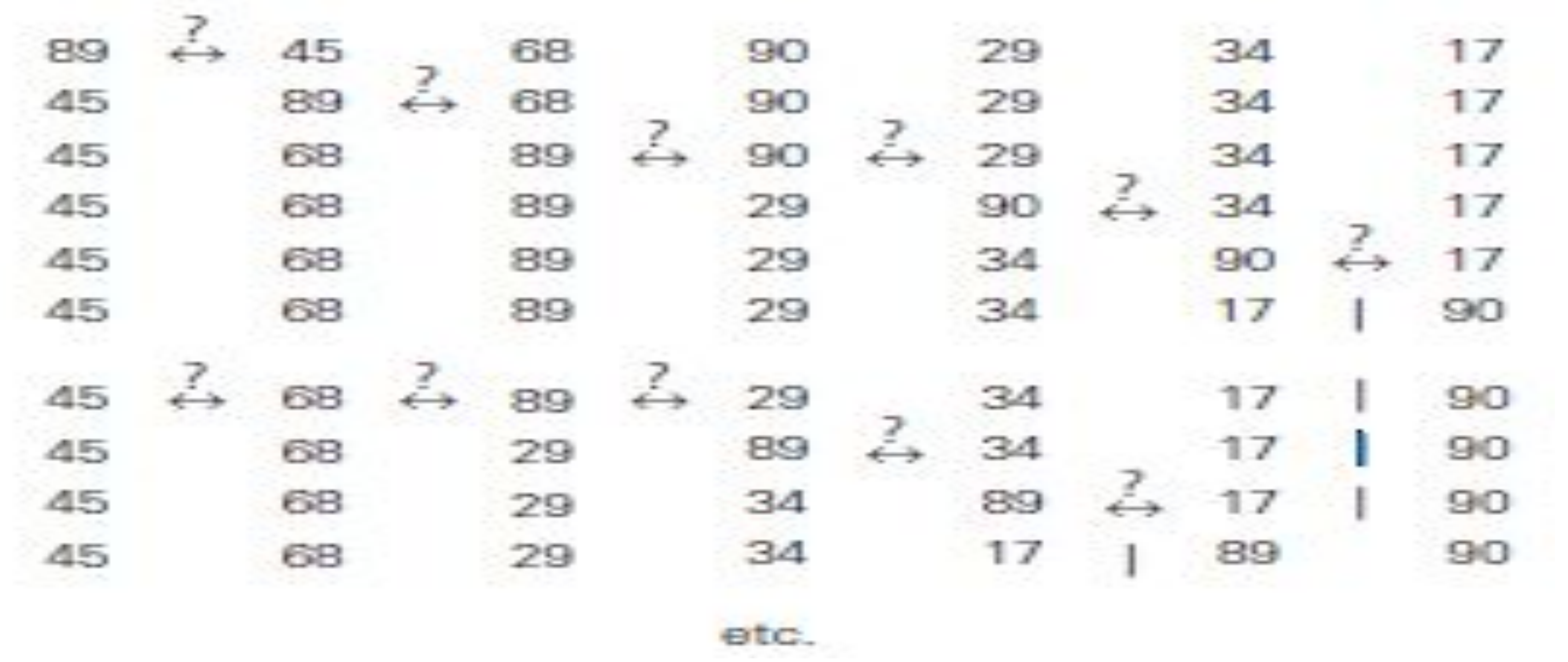
- Scan the array, swapping adjacent pair of elements if they are out of order. This bubbles up the largest element to the end.
- Scan the array again, bubbling up the second largest element.
- Repeat until all elements are in order.

Bubble Sort

- Comparing adjacent elements of the list and exchange them if they are out of order.
- By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list.
- The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted.

Example

- First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17

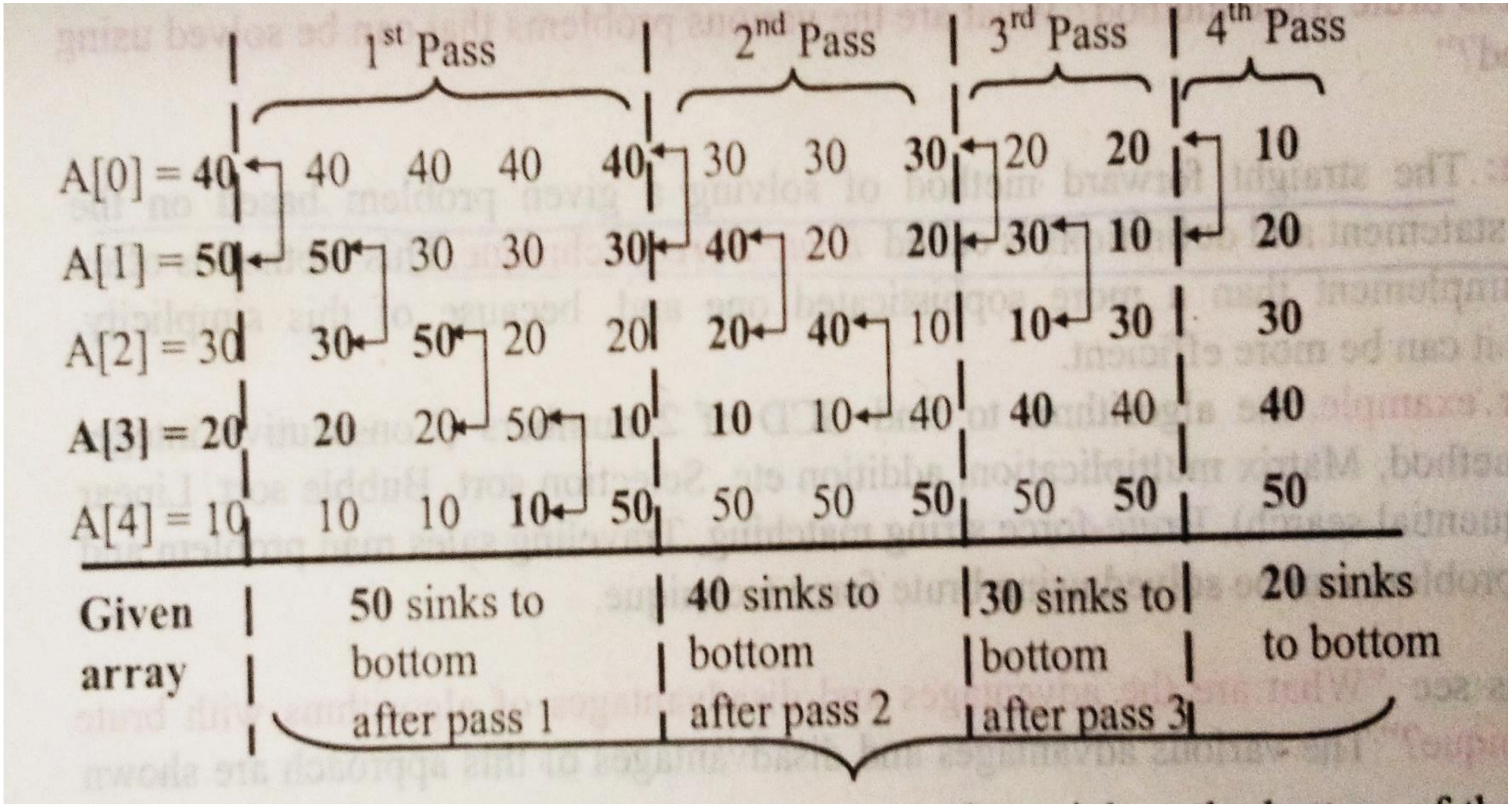


89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

etc.

Example: Bubble Sort

```
for i ←0 to n − 2 {
    for j ←0 to n − 2 − i {
```



Bubble Sort Algorithm

```
ALGORITHM BubbleSort($A[0..n-1]$)
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ to $n-2$ {
    for $j \leftarrow 0$ to $n-2-i$ {
        if $A[j+1] < A[j]$
            swap $A[j]$ and $A[j+1]$
        }
    }
```

Analysis of Bubble Sort Algorithm

- The number of key comparisons for the bubble-sort algorithm is the same for all arrays of size *n*; it is obtained by a sum:

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

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Question

```
ALGORITHM BubbleSort(A[0..n − 1])
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ←0 to n − 2 {
    for j ←0 to n − 2 − i {
        if A[j + 1]<A[j ]
            swap A[j ] and A[j + 1]
            }
        }
```

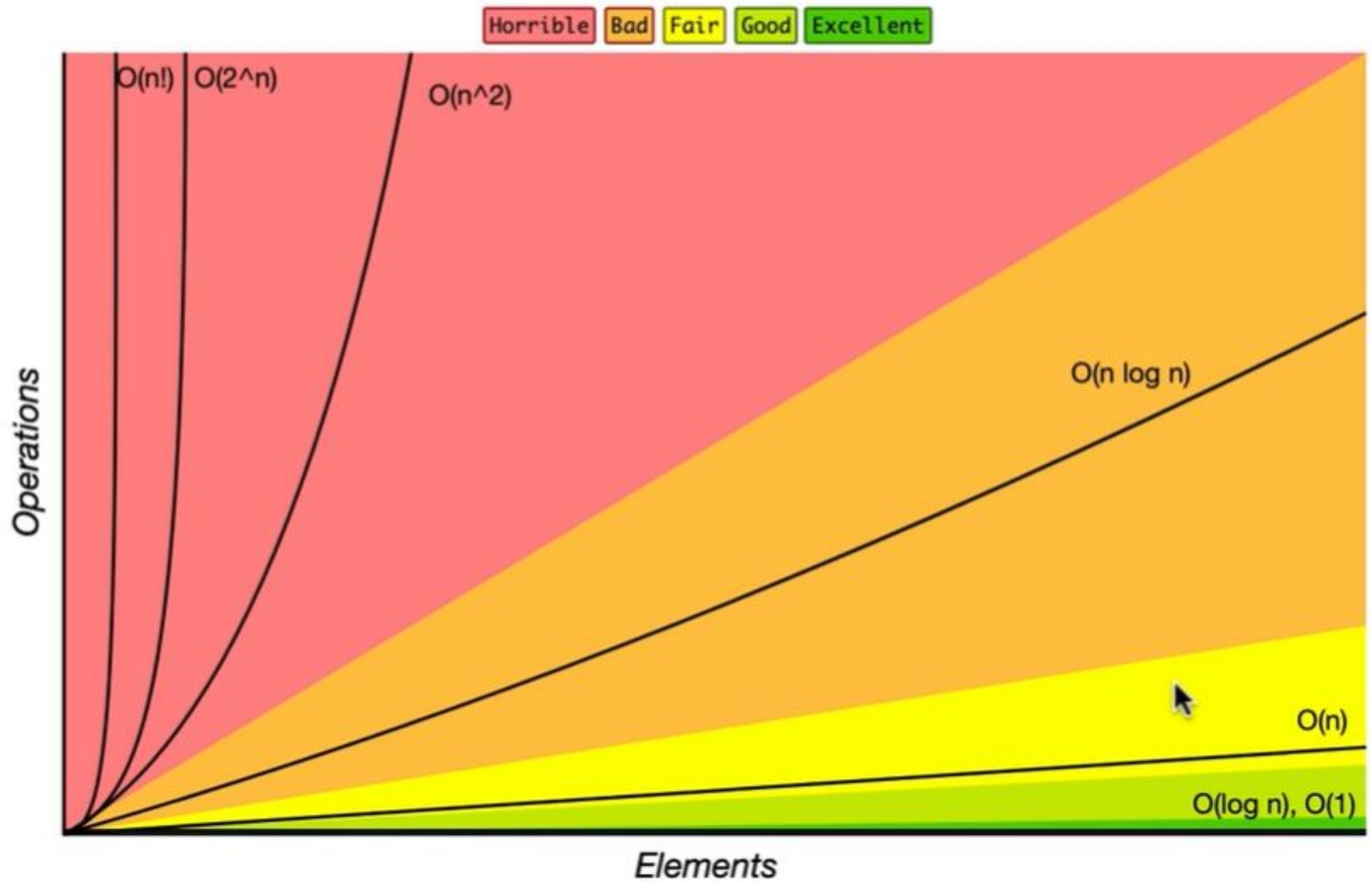
Question:
In the worst case, if the input array given to the algorithm is in decreasing order then How many swap operations will be carried out ?

Answer

- The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Big-O Complexity Chart



Improving Bubble Sort Algorithm

- The first version of an Bubble Sort algorithm obtained can often be improved upon with a modest amount of effort.
- Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm.
- Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases.

Improved Bubble Sort Algorithm

```
ALGORITHM BubbleSort(A[0..n − 1])
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ←0 to n − 2 {
 swapped = false;
     for j ←0 to n − 2 − i {
          if A[j + 1]<A[j ]
               { swap A[j ] and A[j + 1] ; swapped = true;}
                    }
        // IF no two elements were swapped by inner loop, then break
          if (swapped == false)
                    break;
     }
```

Quiz

Assume that we use Improved Bubble Sort to sort n distinct elements in ascending order. When does the best case of Improved Bubble Sort occur?

- ☐ When elements are sorted in ascending order
- ☐ When elements are sorted in descending order
- ☐ When elements are not sorted by any order
- ☐ There is no best case for Improved Bubble Sort. It always takes O(n*n) time

Answer

Assume that we use Improved Bubble Sort to sort n distinct elements in ascending order. When does the best case of Improved Bubble Sort occur?

- **When elements are sorted in ascending order**
- When elements are sorted in descending order
- When elements are not sorted by any order
- There is no best case for Improved Bubble Sort. It always takes O(n*n) time

Question

- The given array is arr = {1,2,3,4,5}. (bubble sort is implemented with a flag variable). The number of iterations in selection sort and bubble sort respectively are,

a) 5 and 4
b) 1 and 4
c) 0 and 4
d) 4 and 1

Answer

- The given array is arr = {1,2,3,4,5}. (bubble sort is implemented with a flag variable i.e., improved Bubble sort). The number of iterations in selection sort and bubble sort respectively are,

a) 5 and 4
b) 1 and 4
c) 0 and 4
d) 4 and 1

Answer: d
Explanation: Selection sort is insensitive to input, hence 4 iterations. Whereas bubble sort iterates only once to set the flag to 0 as the input is already sorted.

Quiz

What is the best time complexity of Improved bubble sort?

- ☐ N^2
- ☐ NlogN
- ☐ N
- ☐ N(logN)^2

Answer

What is the best time complexity of Improved bubble sort?

- N^2
- NlogN
- [✓ **N**]
- N(logN)^2

Question

Following process depicts which sorting algorithm:

First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position and continue in this way until the entire array is sorted.

- ☐ Selection Sort
- ☐ Bubble Sort

Answer

Following process depicts which sorting algorithm:

First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position and continue in this way until the entire array is sorted.

- ☐ Selection Sort
- ☐ Bubble Sort

Question

For each i from 1 to n-1, there are ____ exchanges for selection sort

- ☐ 1
- ☐ n-1
- ☐ n

Answer

For each i from 1 to n-1, there are ____ exchanges for selection sort

- ☐ 1
- ☐ n-1
- ☐ n

Question

A selection sort compares adjacent elements and swaps them if they are in the wrong order

- ☐ True
- ☐ False
- ☐ Depends on the Array Elements

Answer

A selection sort compares adjacent elements and swaps them if they are in the wrong order

- ☐ True
- ☐ False
- ☐ Depends on the Array Elements

Improved Bubble Sort

Improved Bubble Sort

- **Worst and Average Case Time Complexity:** $\Theta(n*n)$. Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:** $O(n)$. Best case occurs when array is already sorted.

Lab Program 3

Sort a given set of N integer elements using Selection Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

To Do

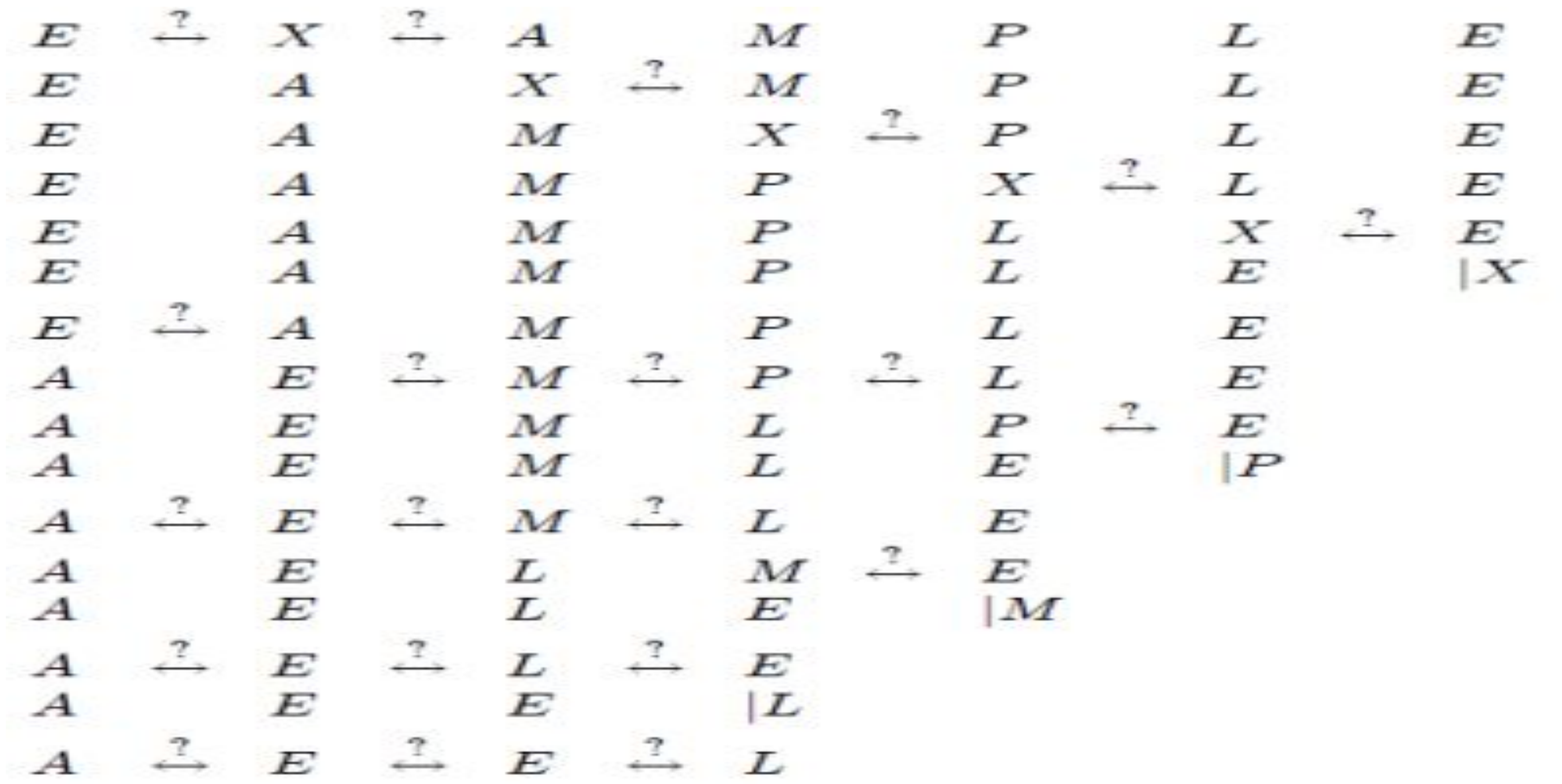
Write C program to sort the elements using Selection sort and Improved Bubble sort for ascending order

- Provide the input array in ascending order
- Execute the program for n
  =10,000
  =15,000
  =20,000
  =25,000

- Plot the graph between **n** and **time taken** for both Selection Sort and Bubble Sort.

Question

- Sort the list “E X A M P L E” in alphabetical order by bubble sort



E	$\overset{?}{\leftrightarrow}$	X	$\overset{?}{\leftrightarrow}$	A		M		P		L		E
E		A		X	$\overset{?}{\leftrightarrow}$	M		P		L		E
E		A		M		X	$\overset{?}{\leftrightarrow}$	P		L		E
E		A		M		P		X	$\overset{?}{\leftrightarrow}$	L		E
E		A		M		P		L		X	$\overset{?}{\leftrightarrow}$	E
E		A		M		P		L		E		$\|X$
E	$\overset{?}{\leftrightarrow}$	A		M		P		L		E		
A		E	$\overset{?}{\leftrightarrow}$	M	$\overset{?}{\leftrightarrow}$	P	$\overset{?}{\leftrightarrow}$	L		E		
A		E		M		L		P	$\overset{?}{\leftrightarrow}$	E		
A		E		M		L		E		$\|P$		
A	$\overset{?}{\leftrightarrow}$	E	$\overset{?}{\leftrightarrow}$	M	$\overset{?}{\leftrightarrow}$	L		E				
A		E		L		M	$\overset{?}{\leftrightarrow}$	E				
A		E		L		E		$\|M$				
A	$\overset{?}{\leftrightarrow}$	E	$\overset{?}{\leftrightarrow}$	L	$\overset{?}{\leftrightarrow}$	E						
A		E		E		$\|L$						
A	$\overset{?}{\leftrightarrow}$	E	$\overset{?}{\leftrightarrow}$	E	$\overset{?}{\leftrightarrow}$	L						

Brute Force Technique

Sorting
- Selection Sort
- Bubble Sort

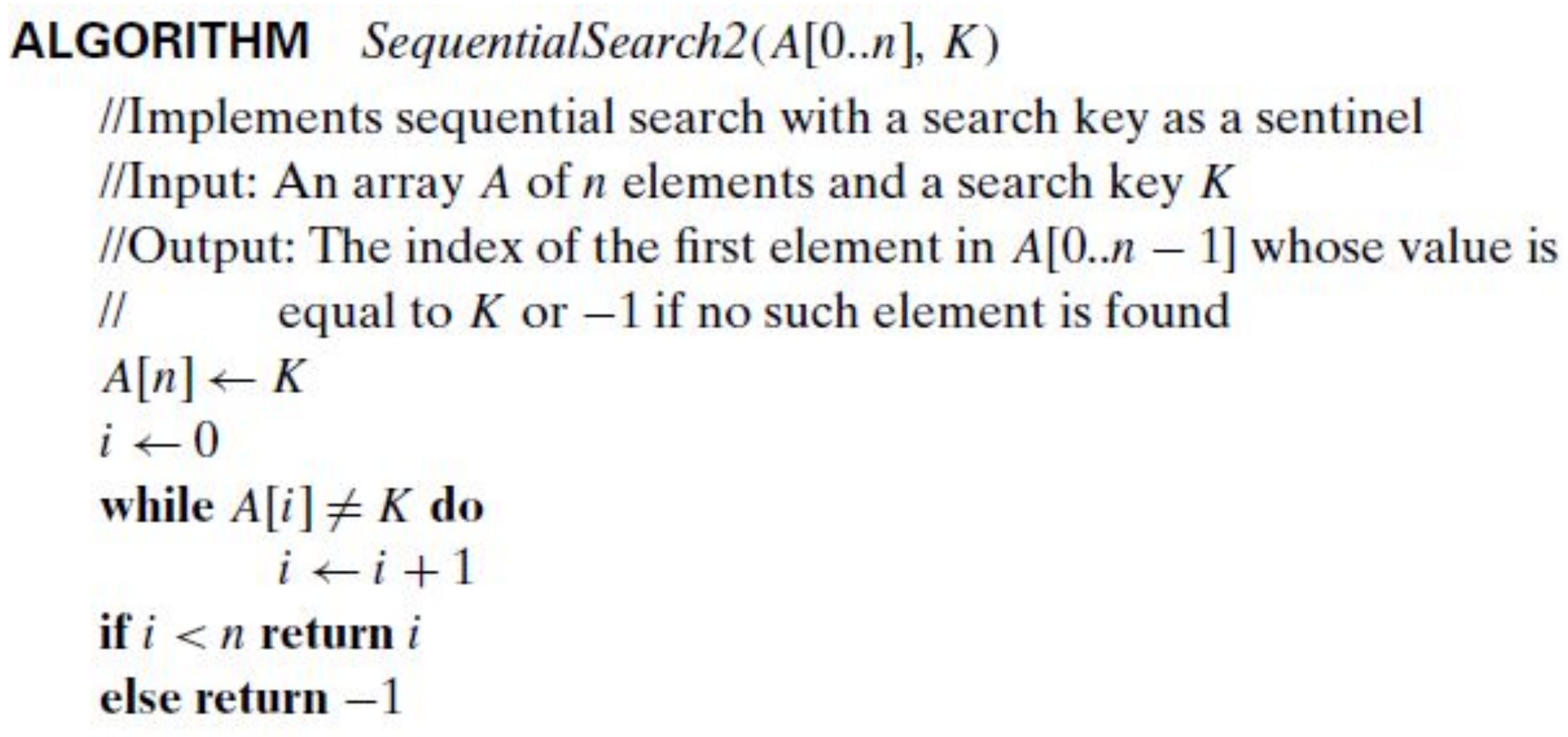
Search
- Sequential Search
- String-Matching problem (or String pattern search)

Brute Force Technique

Search
- Sequential Search

Brute-Force: Sequential Search

- The algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).



```
ALGORITHM SequentialSearch2($A[0..n]$, $K$)
    //Implements sequential search with a search key as a sentinel
    //Input: An array $A$ of $n$ elements and a search key $K$
    //Output: The index of the first element in $A[0..n-1]$ whose value is
    //        equal to $K$ or $-1$ if no such element is found
    $A[n] \leftarrow K$
    $i \leftarrow 0$
    while $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

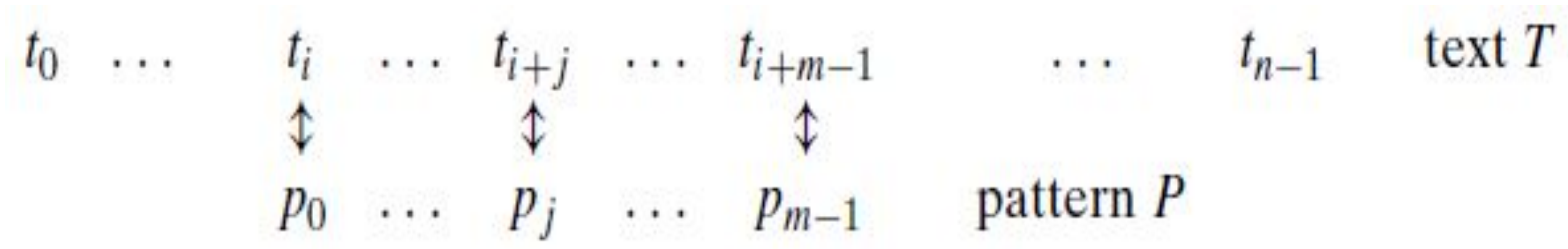
Brute Force Technique

Search

- String-Matching problem (or String pattern search)

Brute Force: String Matching Problem

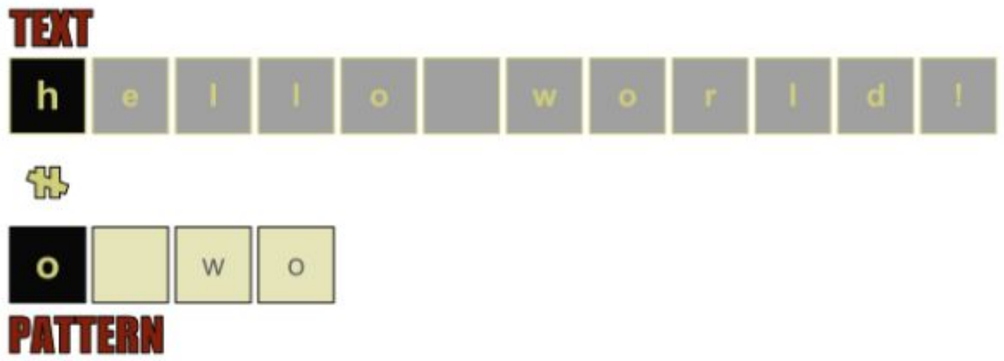
- Given a string of *n* characters called the ***text*** and a string of *m* characters *(m ≤ n)* called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that


$t_0 \quad \dots \quad t_i \quad \dots \quad t_{i+j} \quad \dots \quad t_{i+m-1} \quad \dots \quad t_{n-1}$ text T
$p_0 \quad \dots \quad p_j \quad \dots \quad p_{m-1}$ pattern P

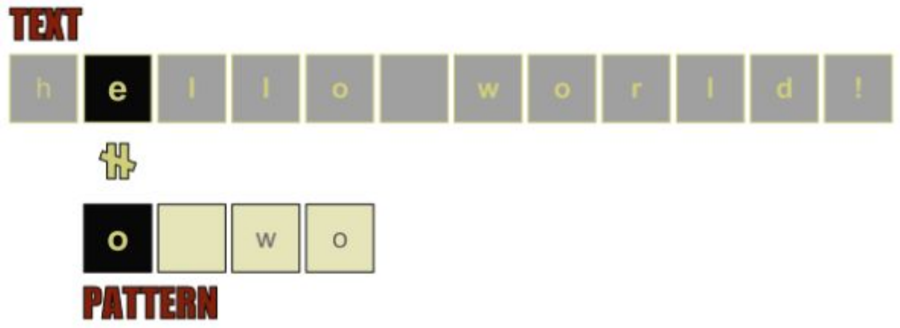
- A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first *m* characters of the text and start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is *n − m*(provided the text positions are indexed from 0 to *n* − 1).

Example: String Matching Problem

- We start by comparing the first characters of the text and the pattern!

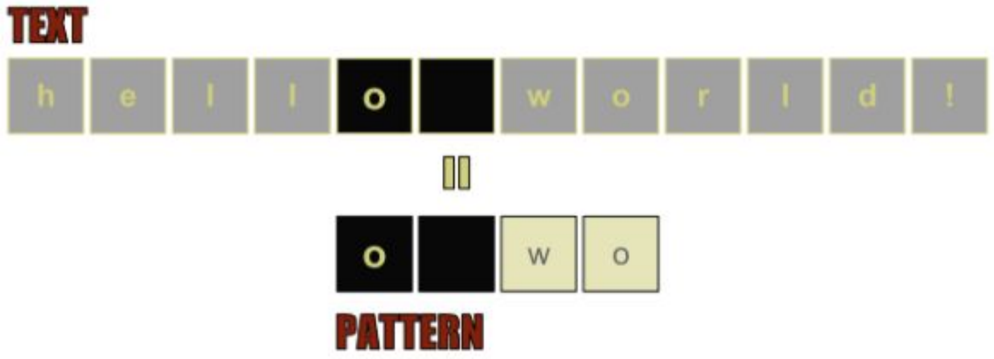


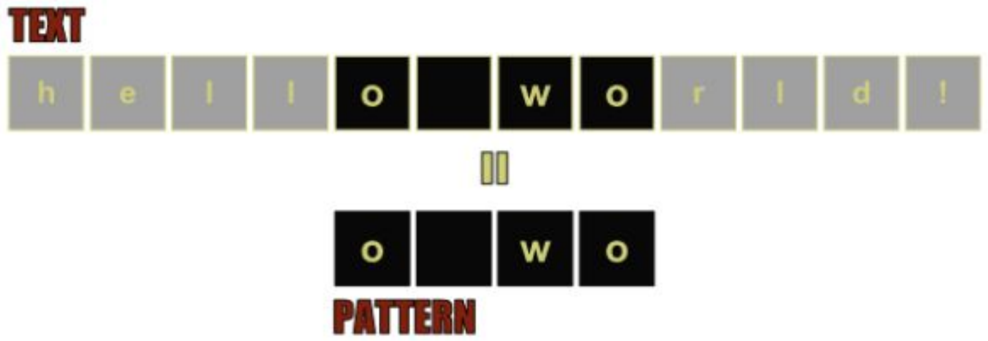
- *Because the first character of the text and the pattern don't match, we move forward the second character of the text. Now we compare the second character of the text with the first character of the pattern!*



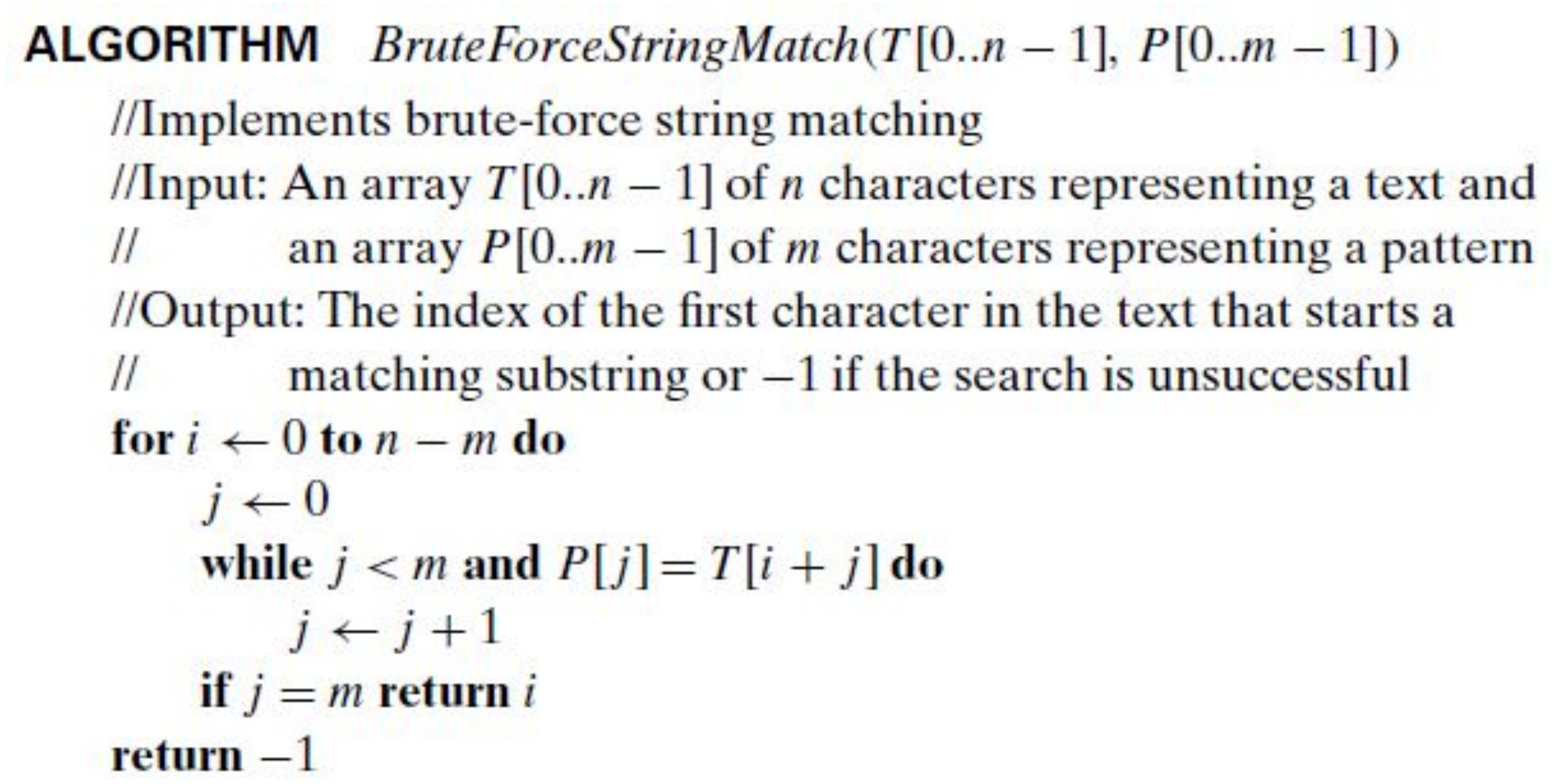
Example: String Matching Problem

- *In case a character from the text match against the first character of the pattern we move forward to the second character of the pattern and the next character of the text!*





String Matching Problem: Algorithm



```
ALGORITHM  BruteForceStringMatch($T[0..n-1]$, $P[0..m-1]$)
    //Implements brute-force string matching
    //Input: An array $T[0..n-1]$ of $n$ characters representing a text and
    //        an array $P[0..m-1]$ of $m$ characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //        matching substring or $-1$ if the search is unsuccessful
    for $i \leftarrow 0$ to $n-m$ do
        $j \leftarrow 0$
        while $j < m$ and $P[j] = T[i+j]$ do
            $j \leftarrow j+1$
        if $j = m$ return $i$
    return $-1$
```

- Trace algorithm for Text="COMPUTER" and Pattern="PUT"

String Matching Problem: Algorithm

Time Analysis

- The algorithm shifts the pattern almost always after a single character comparison.
- The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.

Question

Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GANDHI in the text

THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED

(Assume that the length of the text is 47 characters long – is known before the search starts.)

Question

Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GANDHI in the text THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED (Assume that the length of the text–it is 47 characters long–is known before the search starts.)

Answer:

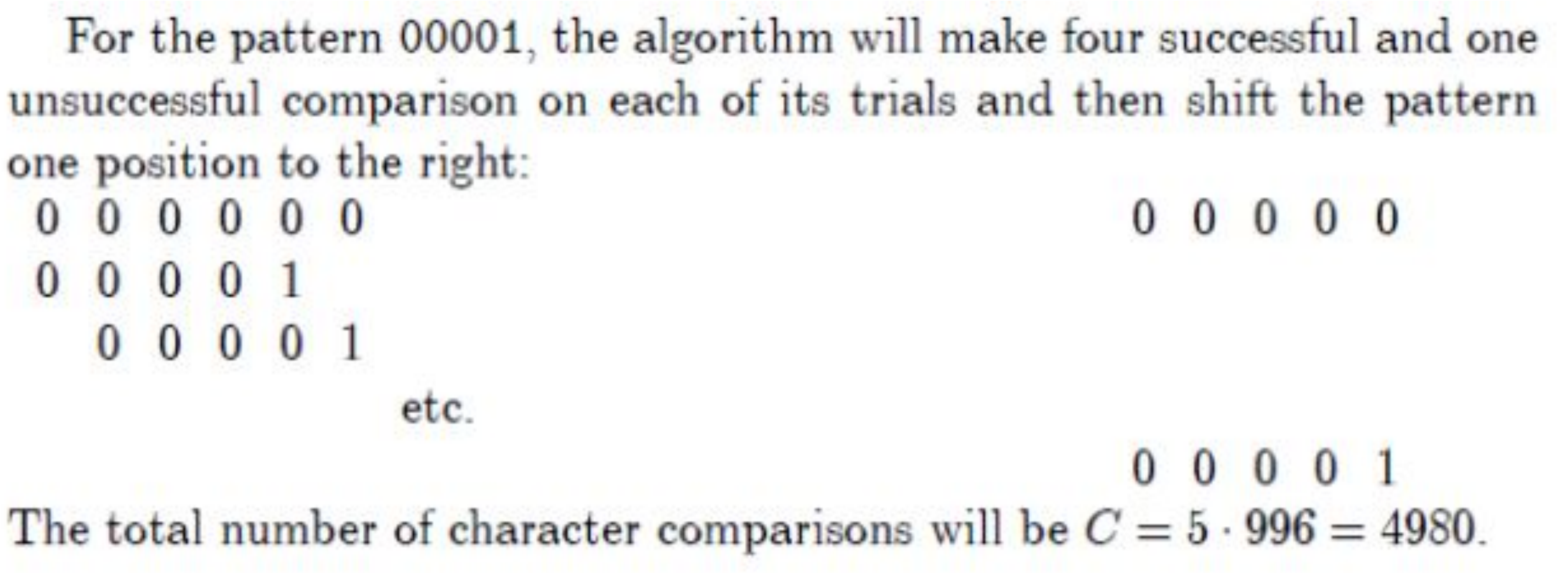
- 43 comparisons. The algorithm will make 47 − 6 + 1 = 42 trials: In the first one, the G of the pattern will be aligned against the first T of the text; in the last one, it will be aligned against the last space. On each but one trial, the algorithm will make one unsuccessful comparison; on one trial–when the G of the pattern is aligned against the G of the text –it will make two comparisons. Thus, the total number of character comparisons will be 41 · 1+ 1 ·2 = 43.

Question

- How many comparisons (both successful and unsuccessful) are made by the brute-force string matching algorithm in searching for pattern "00001" in the binary text of 1000 zeros?
- 00000000000000000000000000000 …0000000
- 00001

Question

- How many comparisons (both successful and unsuccessful) are made by the brute-force string matching algorithm in searching for pattern "00001" in the binary text of 1000 zeros?



For the pattern 00001, the algorithm will make four successful and one unsuccessful comparison on each of its trials and then shift the pattern one position to the right:

```
0 0 0 0 0 0                      0 0 0 0 0
0 0 0 0 1
  0 0 0 0 1
          etc.
                                 0 0 0 0 1
```

The total number of character comparisons will be $C = 5 \cdot 996 = 4980$.

Question

- How many comparisons (both successful and unsuccessful) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

b. 10000 c. 01010

Answer

- How many comparisons (both successful and unsuccessful) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

b. 10000 c. 01010

b) 10000 There will be a total of 1000-5+1 = 996 iterations. In each of these iterations, the first comparison would itself be unsuccessful. Hence, there will be 996*1 = 996 unsuccessful comparisons and there will not be any successful comparisons. Total comparisons = 996.

c) 01010 There will be a total of 1000-5+1 = 996 iterations. In each of these iterations, the first comparison would be successful and the second comparison would be unsuccessful. Hence, there will be 996*1 = 996 successful comparisons and another 996*1 = 996 unsuccessful comparisons. Total comparisons = 1992.

Exhaustive Search

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem

Exhaustive Search

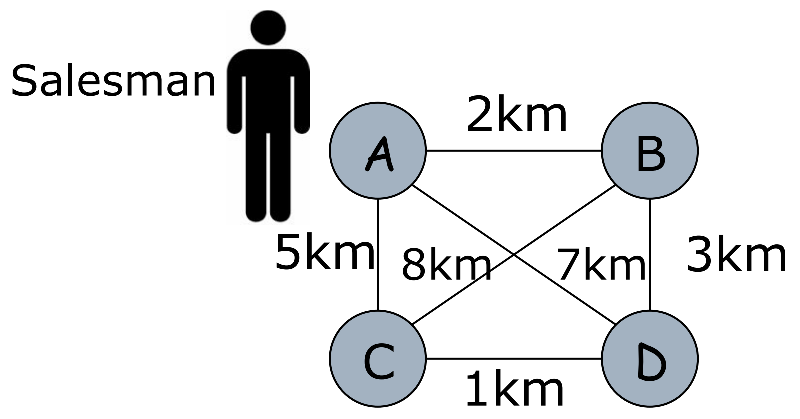
- ***Exhaustive search*** is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element

Exhaustive Search

- Traveling Salesman Problem

Traveling Salesman Problem (TSP)

- S salesman wants to visit all cities, A, B, C and D. What is the best way to do this (Minimal travel distance)?

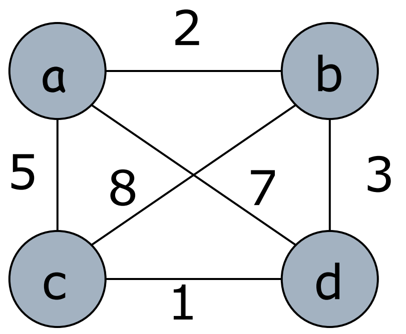


Traveling Salesman Problem (TSP)

- Given a set of cities and distance 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.
- How to Solve TSP ?
  - Get all tours by generating **all permutations of n-1** intermediate cities, compute the tour lengths, and find the shortest among them.

Traveling Salesman Problem (TSP)

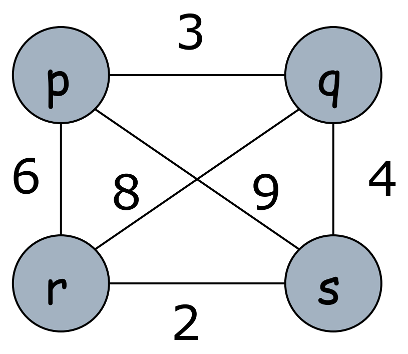
List all possible routes starting from city ‘a’ and find optimal route



a □ b □ c □ d □ a	2+8+1+7 = 18	
a □ b □ d □ c □ a	2+3+1+5 = 11	⟵ **optimal**
a □ c □ b □ d □ a	5+8+3+7 = 23	
a □ c □ d □ b □ a	5+1+3+2 = 11	⟵ **optimal**
a □ d □ b □ c □ a	7+3+8+5 = 23	
a □ d □ c □ b □ a	7+1+8+2 = 18	

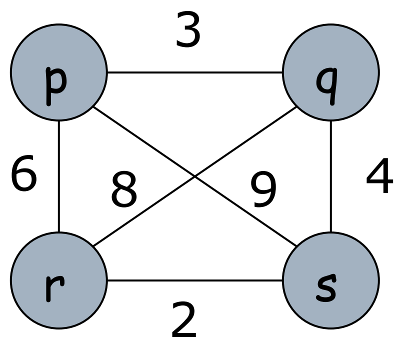
Question

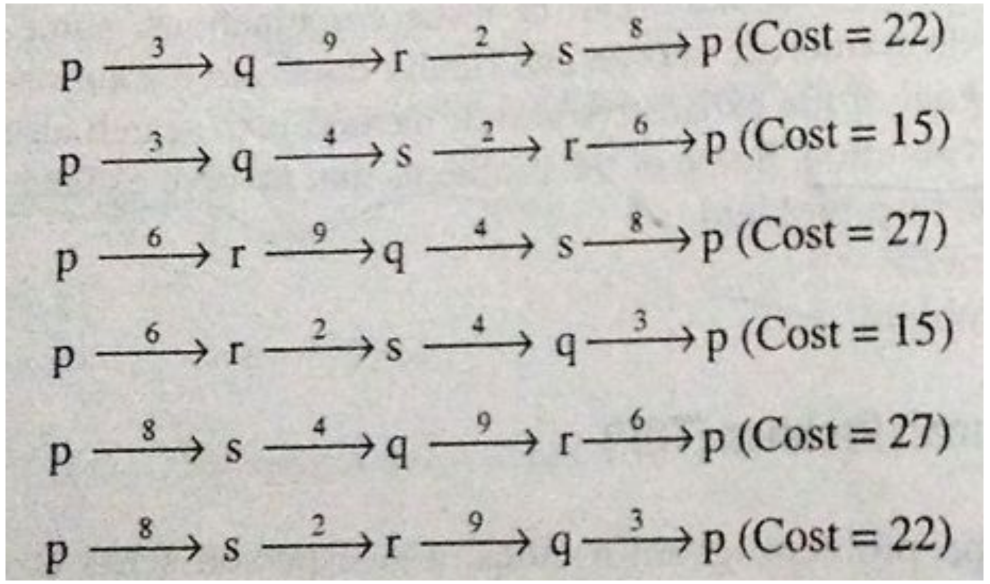
List all tours starting from city p and find the shortest among them



Question

List all tours starting from city p and find the shortest among them





$p \xrightarrow{3} q \xrightarrow{9} r \xrightarrow{2} s \xrightarrow{8} p$ (Cost = 22)

$p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p$ (Cost = 15)

$p \xrightarrow{6} r \xrightarrow{9} q \xrightarrow{4} s \xrightarrow{8} p$ (Cost = 27)

$p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p$ (Cost = 15)

$p \xrightarrow{8} s \xrightarrow{4} q \xrightarrow{9} r \xrightarrow{6} p$ (Cost = 27)

$p \xrightarrow{8} s \xrightarrow{2} r \xrightarrow{9} q \xrightarrow{3} p$ (Cost = 22)

Exhaustive Search: Traveling Salesman Problem

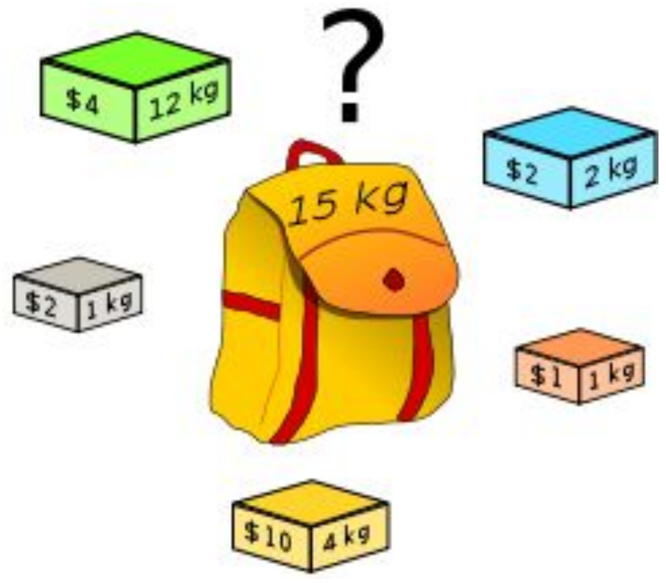
- We can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them.
- Exhaustive-search approach is impractical for all but very small values of n.

Exhaustive Search

- Knapsack Problem

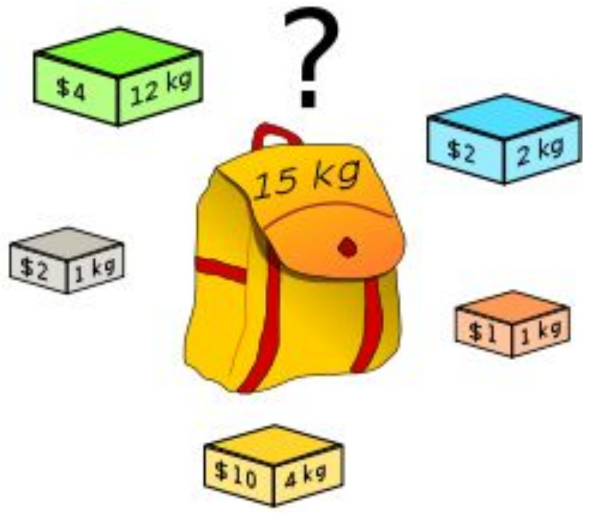
Knapsack Problem

- Given *n* items of known weights $w1, w2, \ldots, w_n$ and values $v1, v2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

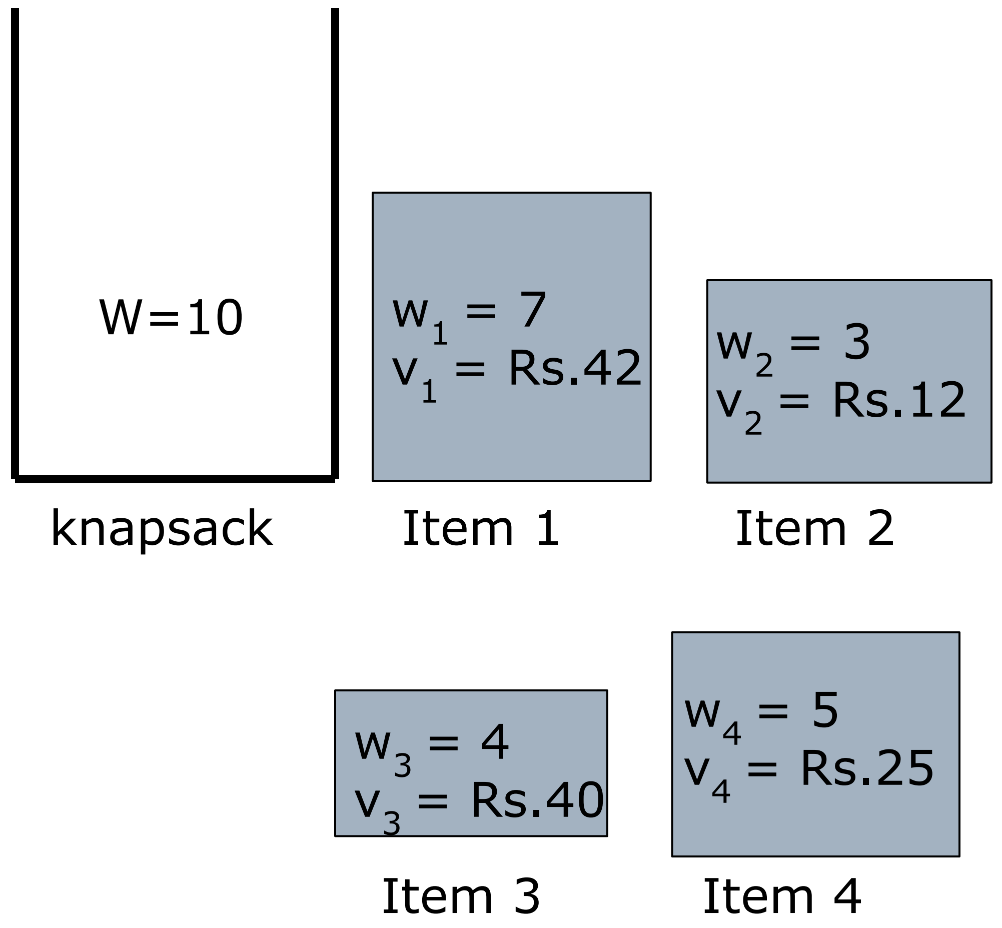


Knapsack Problem

- You have knapsack that has capacity (weight) W
- You have several items $i_1,..i_n$
- Each item i_j has a weight w_j and a value v_j
- You want to place a certain number of each item i_j in the kanpsack so that:
  - The knapsack weight **capacity** is **not exceeded** and
  - The **total value** is **maximal**

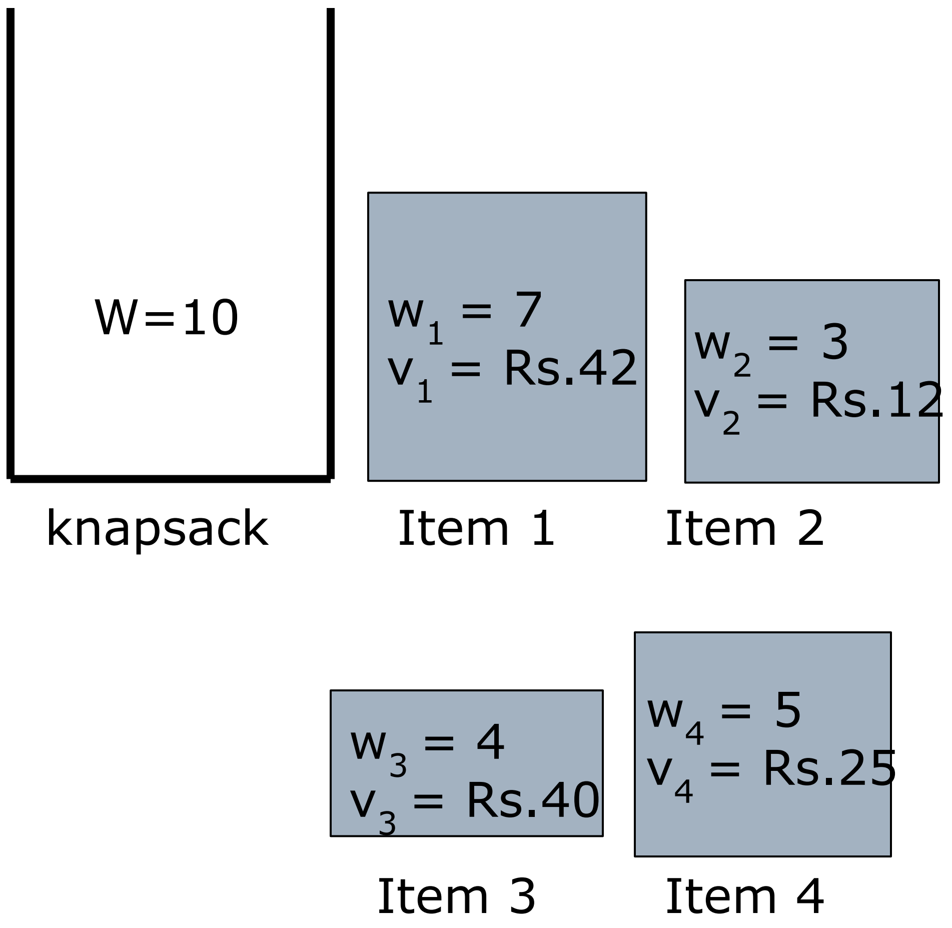


Example



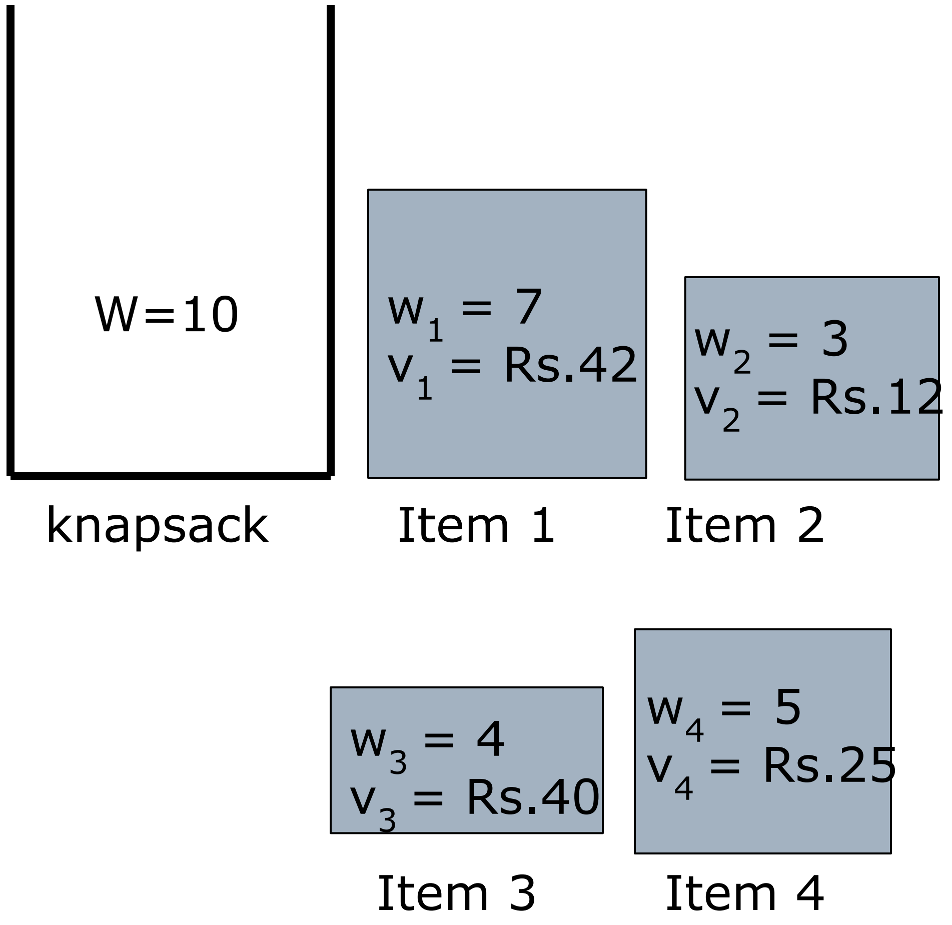
subset	weight	value

Example



subset	weight	value
Ø	0	Rs.0
{1}	7	Rs.42
{2}	3	Rs.12
{3}	4	Rs.40
{4}	5	Rs.25
{1,2}	10	Rs.54
{1,3}	11	Not feasible
{1,4}	12	Not feasible

Example



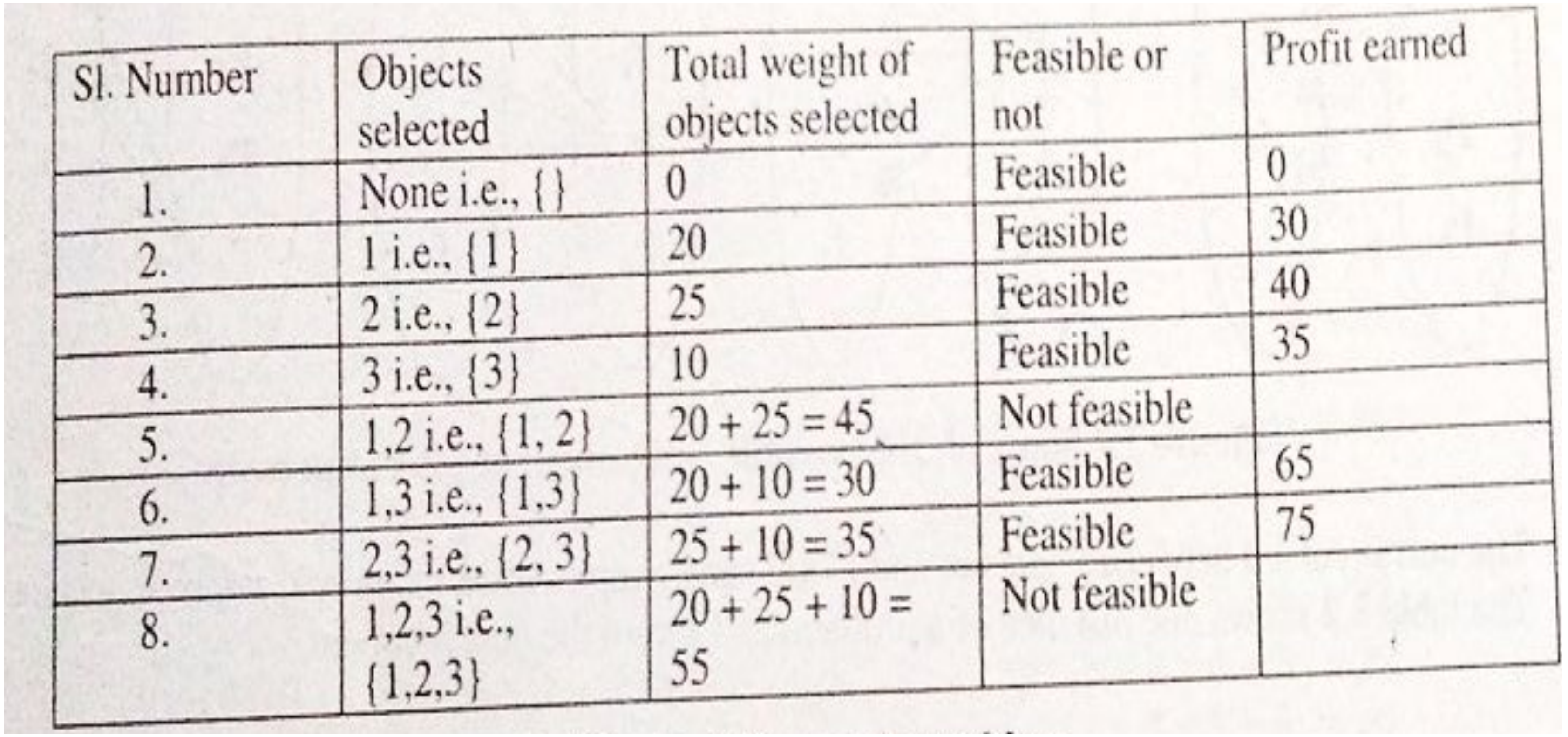
subset	weight	value
Ø	0	Rs.0
{1}	7	Rs.42
{2}	3	Rs.12
{3}	4	Rs.40
{4}	5	Rs.25
{1,2}	10	Rs.54
{1,3}	11	!feasible
{1,4}	12	!feasible
{2,3}	7	Rs.52
{2,4}	8	Rs.37
{3,4}	9	Rs.65
{1,2,3}	14	!feasible
{1,2,4}	15	!feasible
{1,3,4}	16	!feasible
{2,3,4}	12	!feasible
{1,2,3,4}	19	!feasible

Question: Knapsack Problem

- Let W=40. Let the weight of three objects be 20, 25, 10 and value of corresponding objects be 30, 40 and 35. Find the optimal solution.

Question: Knapsack Problem

- Let W=40. Let the weight of three objects be 20, 25, 10 and value of corresponding objects be 30, 40 and 35. Find the optimal solution.



Sl. Number	Objects selected	Total weight of objects selected	Feasible or not	Profit earned
1.	None i.e., { }	0	Feasible	0
2.	1 i.e., {1}	20	Feasible	30
3.	2 i.e., {2}	25	Feasible	40
4.	3 i.e., {3}	10	Feasible	35
5.	1,2 i.e., {1, 2}	20 + 25 = 45	Not feasible	
6.	1,3 i.e., {1,3}	20 + 10 = 30	Feasible	65
7.	2,3 i.e., {2, 3}	25 + 10 = 35	Feasible	75
8.	1,2,3 i.e., {1,2,3}	20 + 25 + 10 = 55	Not feasible	

Exhaustive Search: Knapsack Problem

- The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

- Since the number of subsets of an *n*-element set is $\mathbf{2^n}$, the exhaustive search leads to a 2^n algorithm, no matter how efficiently individual subsets are generated.

Exhaustive Search

- Assignment Problem

Assignment Problem

- There are n people who need to be assigned to execute n jobs, one person per job.
- C[i, j] : cost that would occur if i^{th} person is assigned to j^{th} job.
- Find an assignment with the minimum total cost.

Assignment Problem

- Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

Person A
Takes 8 units
Of time to
Complete Job4

	Job 1	Job 2	Job 3	Job 4
Person A	Rs. 9	Rs. 2	Rs. 7	Rs. 8
Person B	Rs. 6	Rs. 4	Rs. 3	Rs. 7
Person C	Rs. 5	Rs. 8	Rs. 1	Rs. 8
Person D	Rs. 7	Rs. 6	Rs. 9	Rs. 4

Assignment Problem

- Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

Person A
Takes 8 units
Of time to
Complete Job4

	Job 1	Job 2	Job 3	Job 4
Person A	9	2	7	8
Person B	6	4	3	7
Person C	5	8	1	8
Person D	7	6	9	4

Green values show optimal job assignment
that is A-Job2, B-Job1, C-Job3 and D-Job4

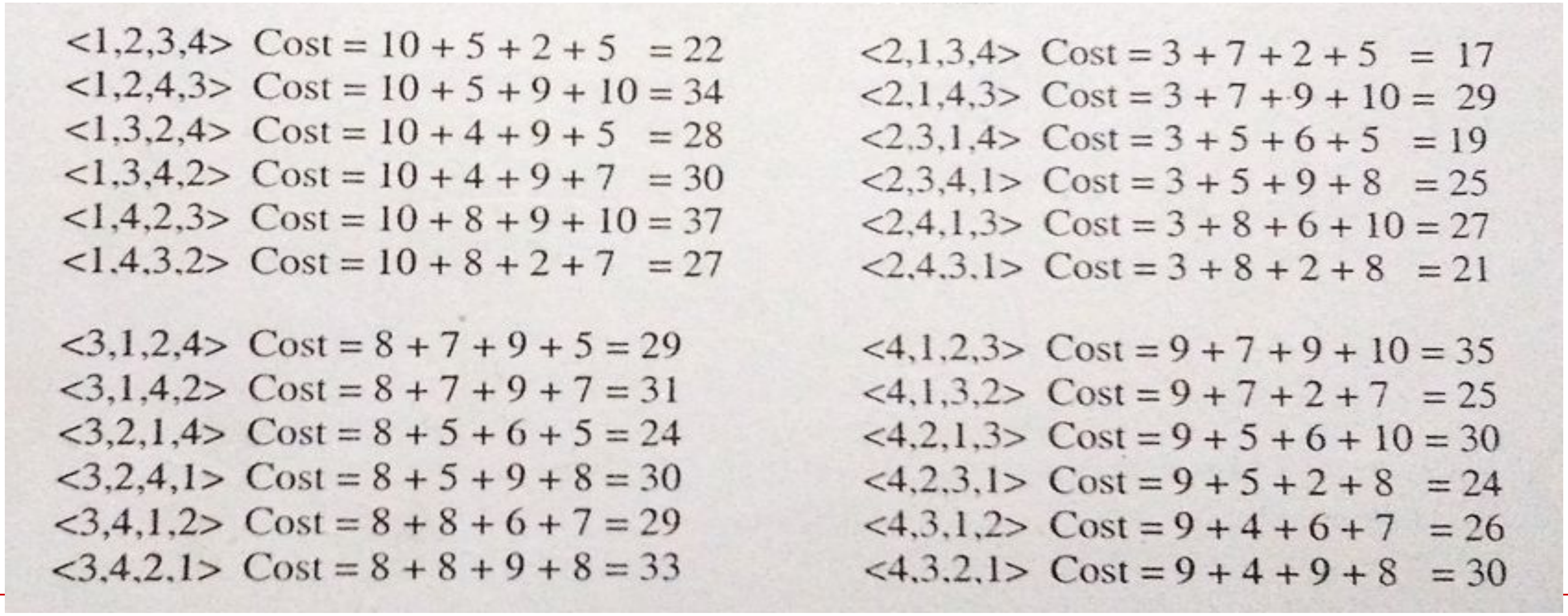
Example: Assignment Problem

Find the optimal job assignment for the following

	Job 1	Job 2	Job 3	Job 4
Person A	10	3	8	9
Person B	7	5	4	8
Person C	6	9	2	9
Person D	8	7	10	5

Example: Assignment Problem

	Job 1	Job 2	Job 3	Job 4
Person A	10	3	8	9
Person B	7	5	4	8
Person C	6	9	2	9
Person D	8	7	10	5



<1,2,3,4> Cost = 10 + 5 + 2 + 5 = 22
<1,2,4,3> Cost = 10 + 5 + 9 + 10 = 34
<1,3,2,4> Cost = 10 + 4 + 9 + 5 = 28
<1,3,4,2> Cost = 10 + 4 + 9 + 7 = 30
<1,4,2,3> Cost = 10 + 8 + 9 + 10 = 37
<1,4,3,2> Cost = 10 + 8 + 2 + 7 = 27

<2,1,3,4> Cost = 3 + 7 + 2 + 5 = 17
<2,1,4,3> Cost = 3 + 7 + 9 + 10 = 29
<2,3,1,4> Cost = 3 + 5 + 6 + 5 = 19
<2,3,4,1> Cost = 3 + 5 + 9 + 8 = 25
<2,4,1,3> Cost = 3 + 8 + 6 + 10 = 27
<2,4,3,1> Cost = 3 + 8 + 2 + 8 = 21

<3,1,2,4> Cost = 8 + 7 + 9 + 5 = 29
<3,1,4,2> Cost = 8 + 7 + 9 + 7 = 31
<3,2,1,4> Cost = 8 + 5 + 6 + 5 = 24
<3,2,4,1> Cost = 8 + 5 + 9 + 8 = 30
<3,4,1,2> Cost = 8 + 8 + 6 + 7 = 29
<3,4,2,1> Cost = 8 + 8 + 9 + 8 = 33

<4,1,2,3> Cost = 9 + 7 + 9 + 10 = 35
<4,1,3,2> Cost = 9 + 7 + 2 + 7 = 25
<4,2,1,3> Cost = 9 + 5 + 6 + 10 = 30
<4,2,3,1> Cost = 9 + 5 + 2 + 8 = 24
<4,3,1,2> Cost = 9 + 4 + 6 + 7 = 26
<4,3,2,1> Cost = 9 + 4 + 9 + 8 = 30

Exhaustive Search: Assignment Problem

- We can describe feasible solutions to the assignment problem as *n*-tuples *j*1, . . . , *jn* in which the *i*th component, *i* = 1, . . . , *n*, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person).
- The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers.
- Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, . . . , *n*, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.
- Since the number of permutations to be considered for the general case of **the assignment problem is *n*!, exhaustive search is impractical for all but very small instances of the problem.**

Depth First Search (DFS) and Breadth First Search (BFS)

- DFS and BFS are the applications of Decrease by one technique

Introduction

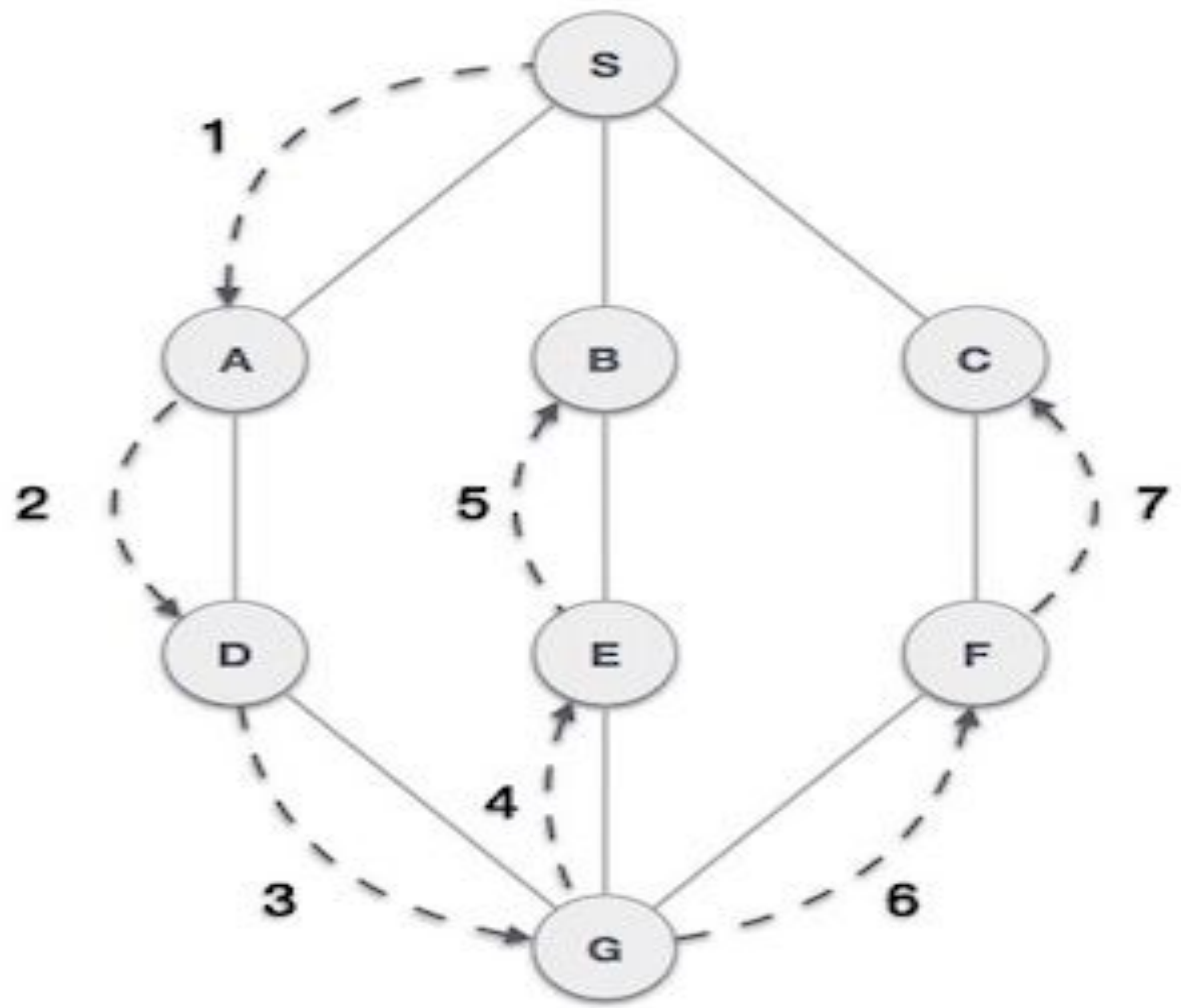
- Many graph algorithms require processing vertices or edges of a graph in systematic fashion. There are two principal algorithms for doing such traversal
  - Depth First Search
  - Breadth First Search

Depth First Search

Depth First Search (DFS)

- Depth first starts visiting vertices of a graph at an arbitrary vertex by marking it as having visited.
- On each iteration, the algorithm proceeds to the unvisited vertex that is adjacent to the one it is currently in.
- The process continues until a dead end – a vertex with no adjacent unvisited vertices is encountered.
- At dead end, the algorithm backs up one edge to the vertex it came from and tries from there.
- The algorithm eventually halts after backing upto the starting vertex, with latter being dead end.

Example



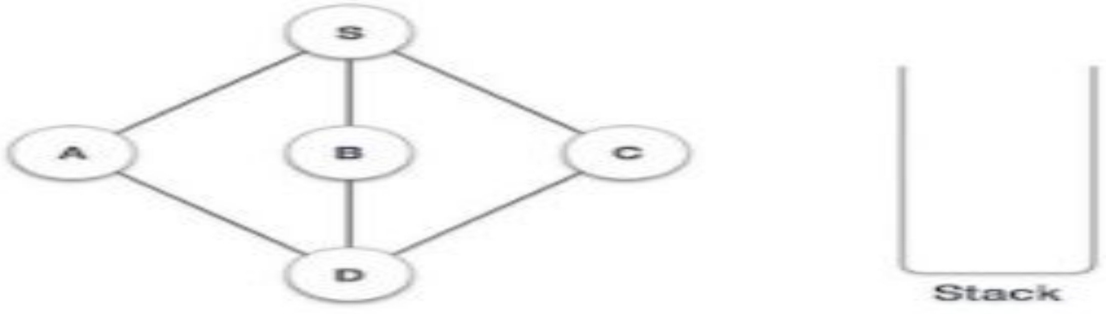
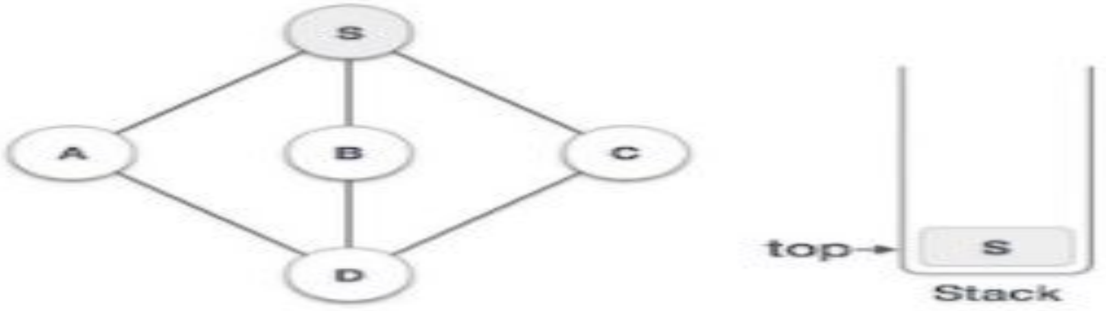
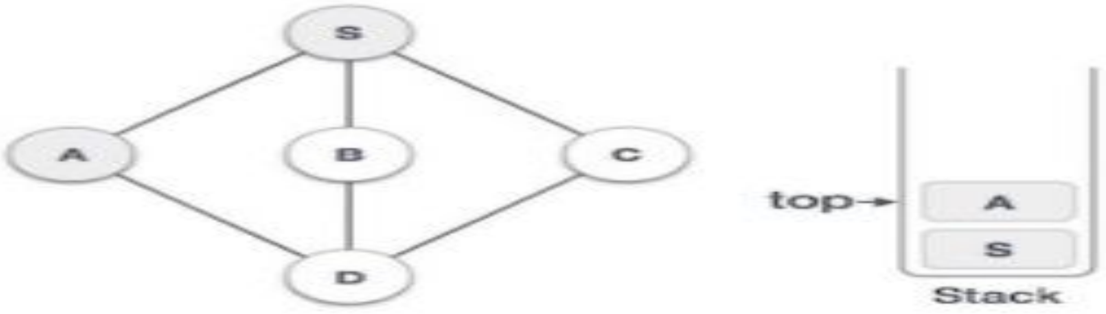
It employs the following rules.
Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

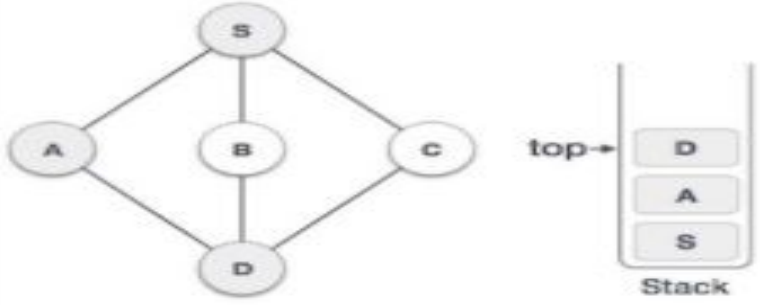
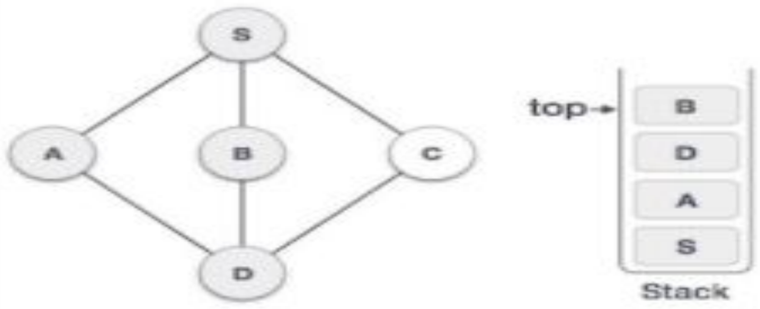
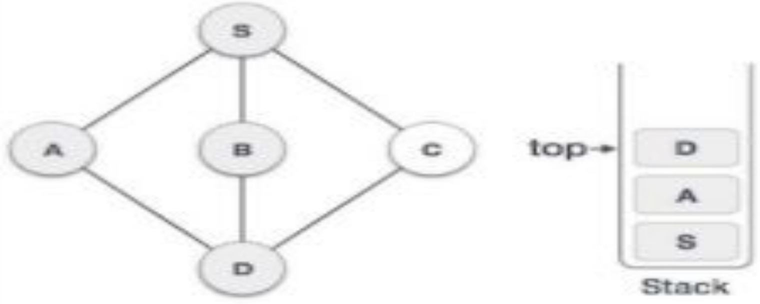
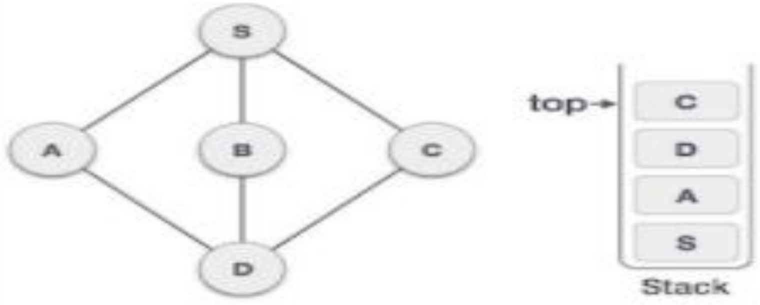
Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.

Example

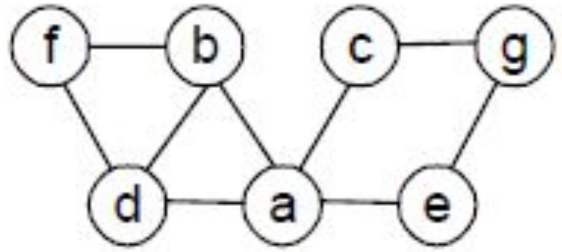
Step	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Question

Consider the graph



$$\begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ \hline a & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ f & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

a	$\to b \to c \to d \to e$
b	$\to a \to d \to f$
c	$\to a \to g$
d	$\to a \to b \to f$
e	$\to a \to g$
f	$\to b \to d$
g	$\to c \to e$

a. Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

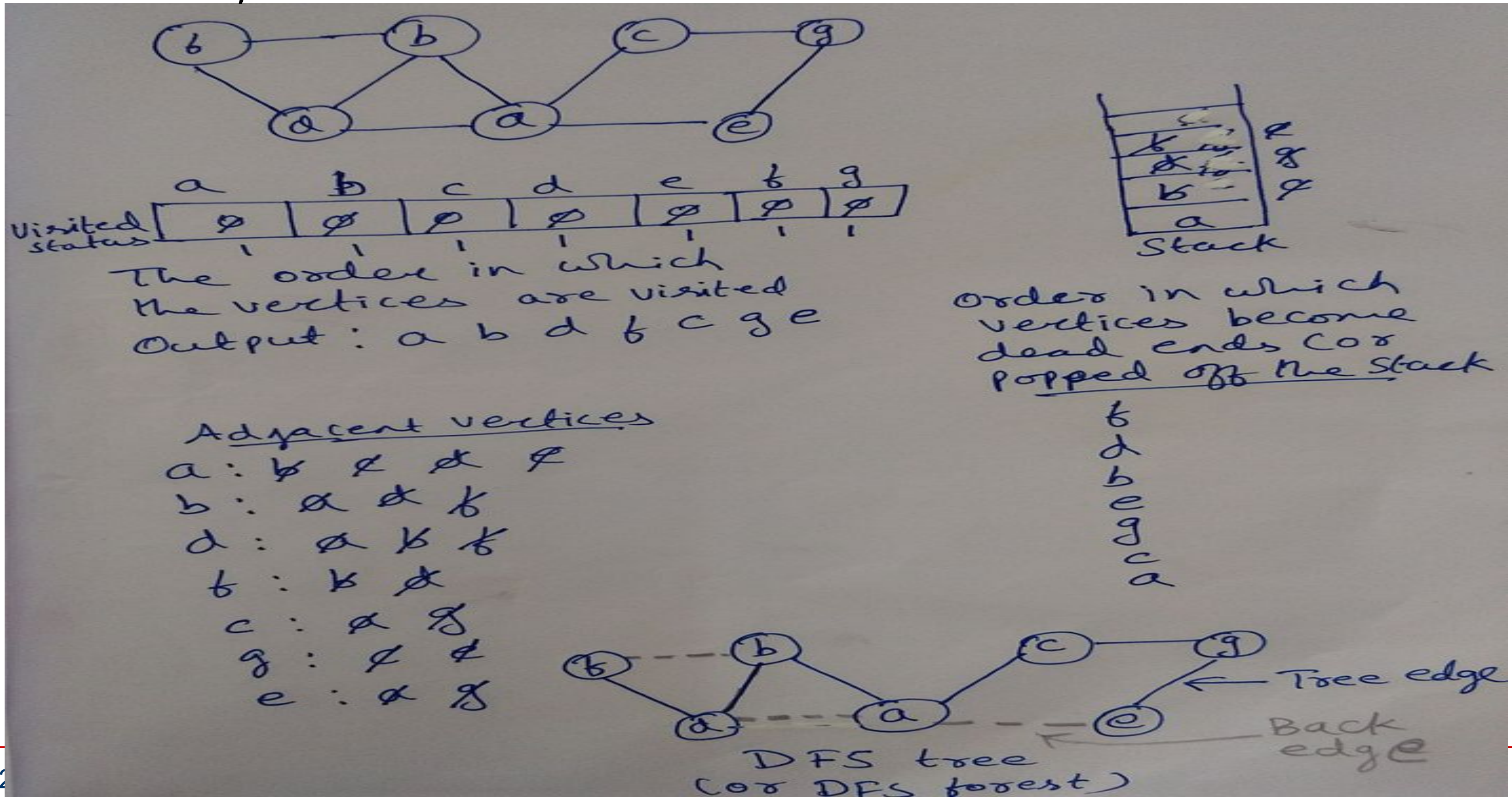
b. Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

Answer

Idea of DFS: To go forward (in depth) while there is any such possibility, if not then, backtrack.

$$\begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ \hline a & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ f & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

$\boxed{a} \to b \to c \to d \to e$
$\boxed{b} \to a \to d \to f$
$\boxed{c} \to a \to g$
$\boxed{d} \to a \to b \to f$
$\boxed{e} \to a \to g$
$\boxed{f} \to b \to d$
$\boxed{g} \to c \to e$



	a	b	c	d	e	f	g
Visited status	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1

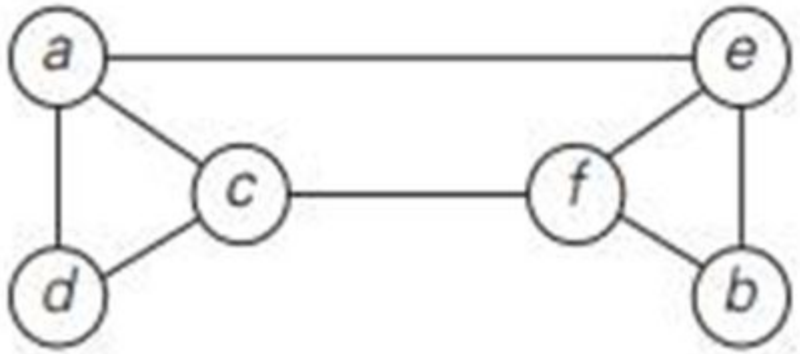
The order in which the vertices are visited
Output : a b d f c g e

Order in which vertices become dead ends (or popped off the stack)
f
d
b
e
g
c
a

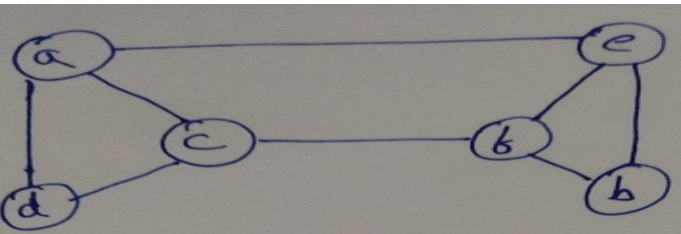
Adjacent vertices
a : ~~b~~ ~~c~~ ~~d~~ ~~e~~
b : ~~a~~ ~~d~~ ~~f~~
d : ~~a~~ ~~b~~ ~~f~~
f : ~~b~~ ~~d~~
c : ~~a~~ ~~g~~
g : ~~c~~ ~~e~~
e : ~~a~~ ~~g~~

DFS tree
(or DFS forest)

Find DFS



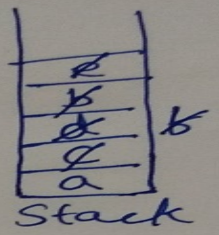
Example



Visited Status	a	b	c	d	e	f
	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1

Order in which vertices are visited

Output: a c d f b e



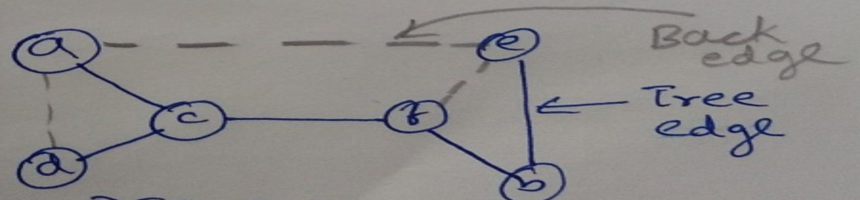
Stack

The order in which nodes become dead ends (or popped off stack)

d
e
b
f
c
a

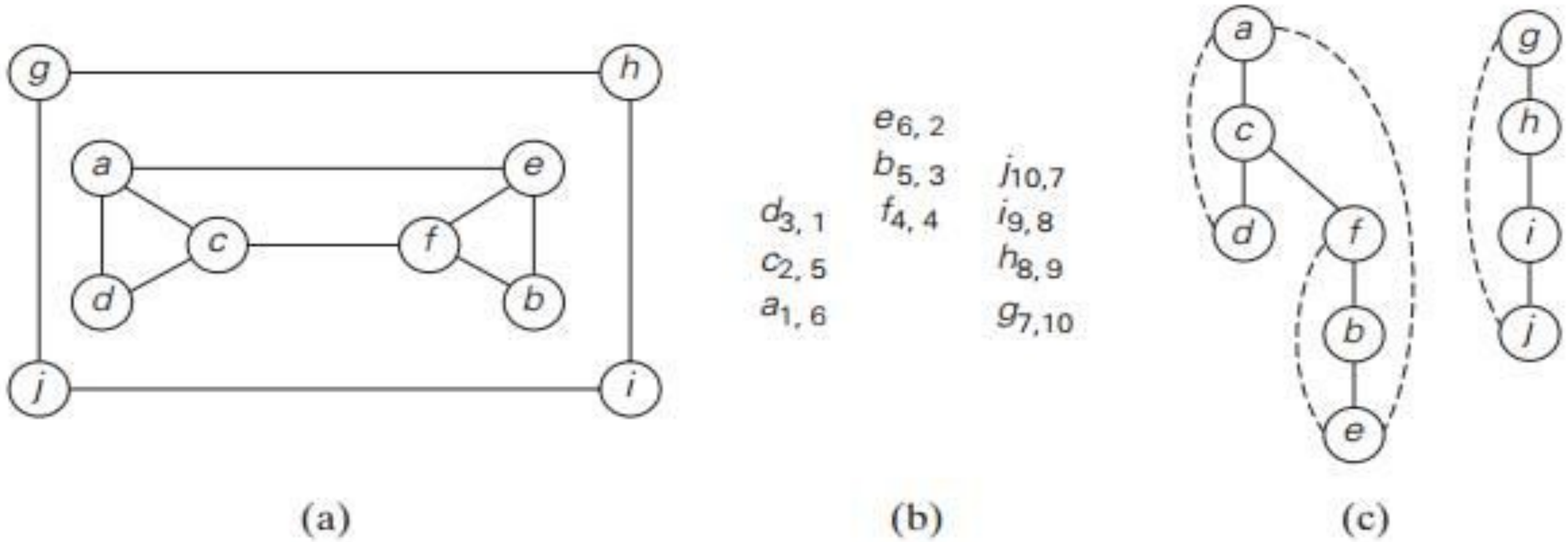
Adjacent vertices

a: ~~c~~ ~~d~~ ~~e~~
c: ~~a~~ ~~d~~ ~~f~~
d: ~~a~~ ~~c~~
f: ~~b~~ ~~c~~ ~~e~~
b: ~~e~~ ~~f~~
e: ~~a~~ ~~b~~ ~~f~~



DFS Tree (or DFS forest)

Example



Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

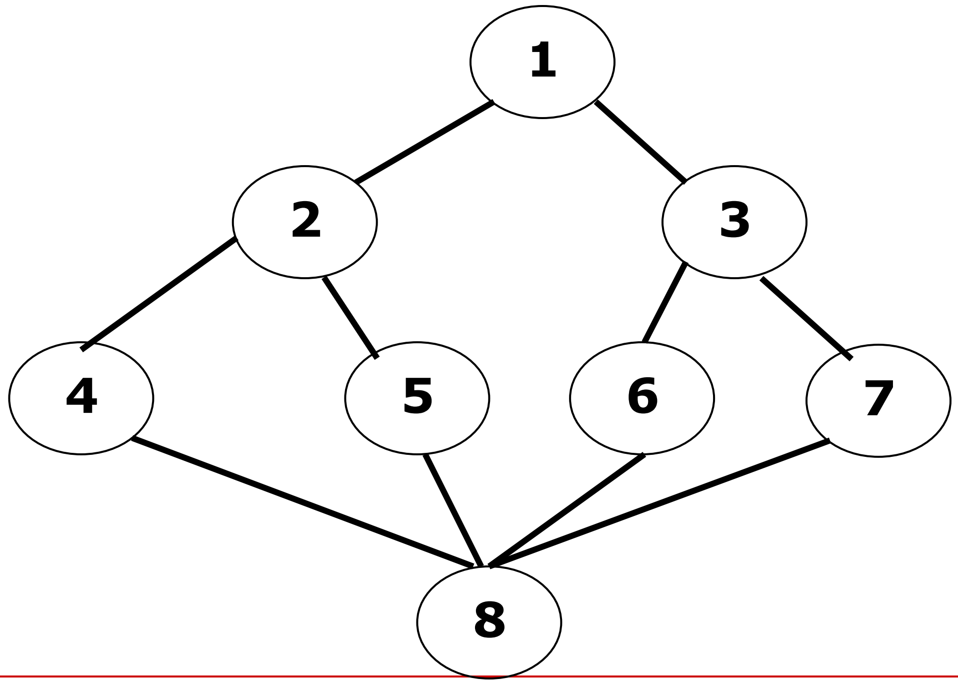
Depth-first search forest

The starting vertex of the traversal serves as the root of the first tree in such a forest.

- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest.
- The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest.

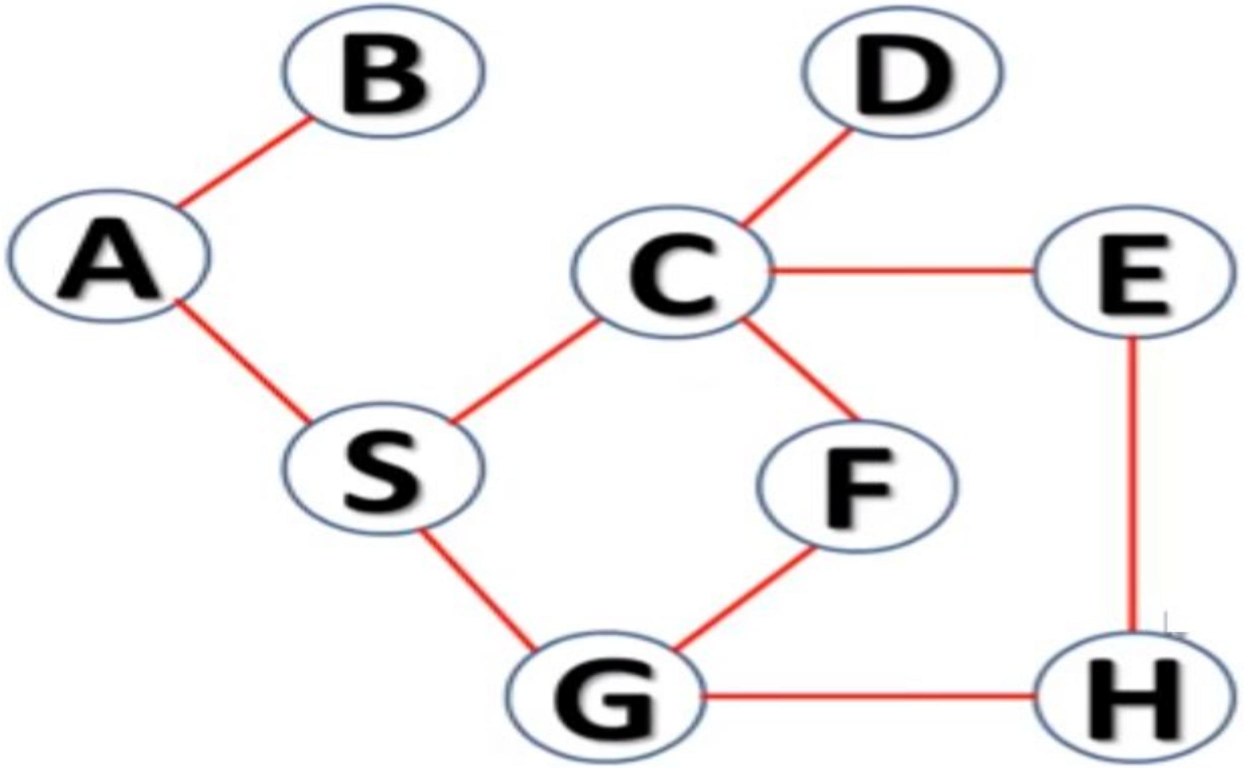
Homework Problem

Starting at vertex 1 and resolving ties by the vertex in ascending order numbering, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).



Homework Problem

- Starting at vertex A and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).



DFS Algorithm

$$\begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ \hline a & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ f & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

```
n ← number of nodes
//Initialize visited[ ] to false (0)
for(i=0;i<n;i++)
    visited[i] = 0;

void DFS(v) //DFS starting from vertex v
{
    print(v);
    visited[v]=1;
    for i=1 to n
     {
        if(adjacent[v][i]==1) and visited[i]==0)
                DFS (i);
          }
}
```

Depth First Search (DFS) Program in C [Adjacency Matrix]

```
#include<stdio.h>
void DFS(int);
int G[10][10],visited[10],n;    //n is no of vertices and graph is sorted in array
G[10][10]
void main()
{
    int i,j;
    printf("Enter number of vertices:");

    scanf("%d",&n);
    //read the adjecency matrix
    printf("\nEnter adjecency matrix of the graph:");

    for(i=0;i<n;i++)
       for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    //visited is initialized to zero
    for(i=0;i<n;i++)
                visited[i]=0;
    DFS(0);
}
```

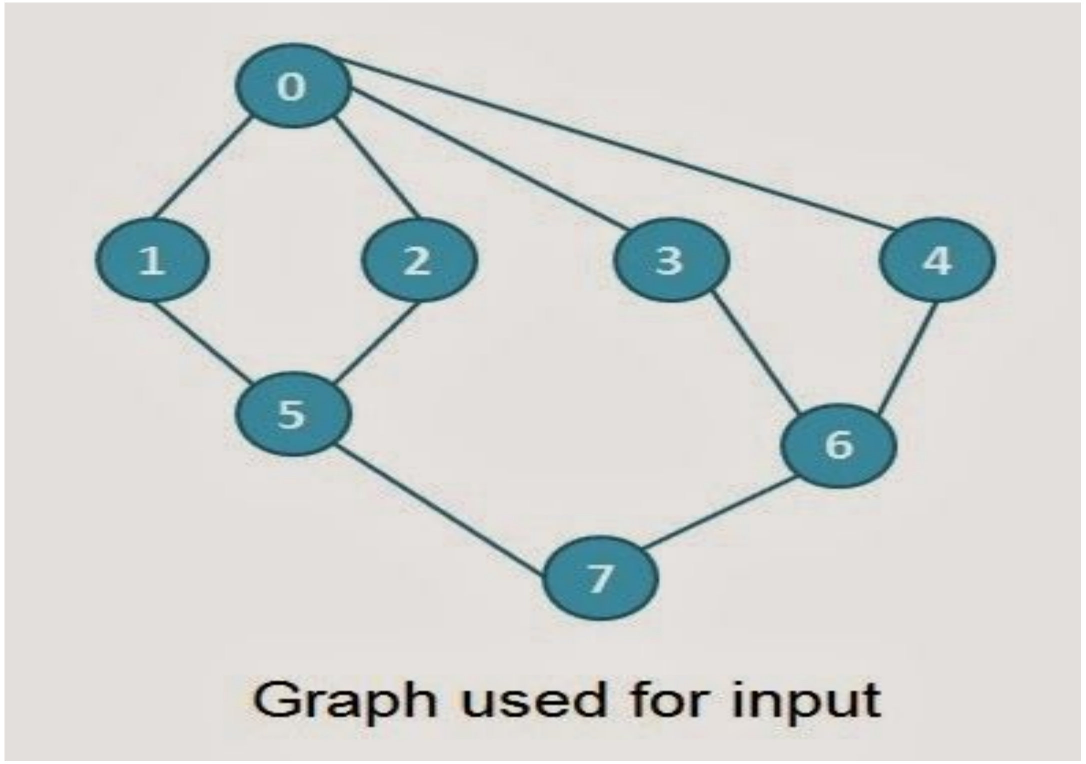
$$
\begin{array}{c|ccccccc}
 & a & b & c & d & e & f & g \\
\hline
a & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
b & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
c & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
d & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
e & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
f & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
g & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

Depth First Search (DFS) Program in C [Adjacency Matrix]

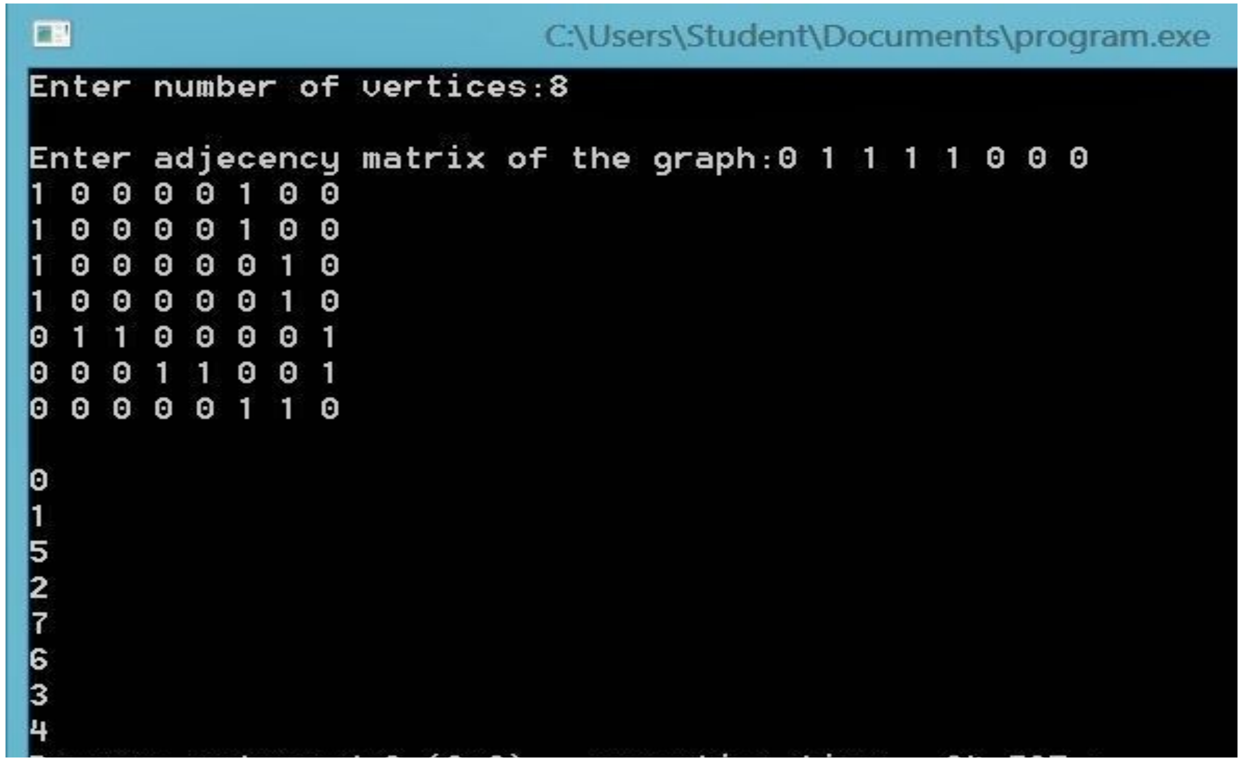
```
void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;

    for(j=0;j<n;j++)
       if(!visited[j] && G[i][j]==1)
            DFS(j);
}
```

$$\begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ \hline a & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ f & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Output



Graph used for input



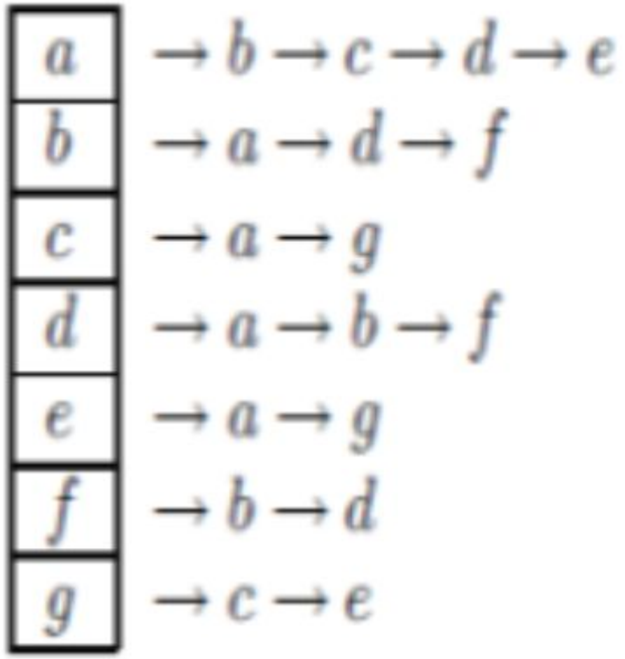
```
Enter number of vertices:8

Enter adjecency matrix of the graph:0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0
0 1 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 0 1 1 0

0
1
5
2
7
6
3
4
```

The graph shown above is taken as input for the program

Depth First Search (DFS) Program in C [Adjacency List]

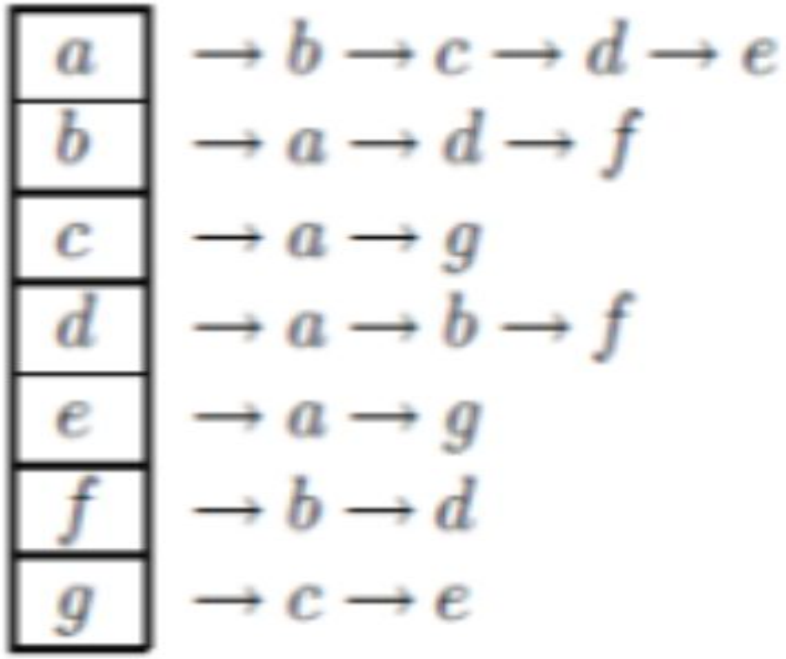
```
#include<stdio.h>
#include<stdlib.h>
 typedef struct node
{
    struct node *next;
    int vertex;
}node;
node *G[20];
//heads of linked list
int visited[20]; int n;
void read_graph();
//create adjacency list
void insert(int,int);
//insert an edge (vi,vj) in
the adjacency list
void DFS(int);
```

```
void main()
{
    int i;
    read_graph();
    //initialised visited to 0

    for(i=0;i<n;i++)
        visited[i]=0;
    DFS(0);
}
```



Depth First Search (DFS) Program in C [Adjacency List]

```
void DFS(int i)
{
   node *p;

   printf("\n%d",i);
   p=G[i];
   visited[i]=1;
   while(p!=NULL)
   {
     i=p->vertex;

     if(!visited[i])
         DFS(i);
      p=p->next;
   }
}
```



Depth First Search (DFS) Program in C [Adjacency List]

```
void read_graph()
{
    int i,vi,vj,no_of_edges;
    printf("Enter number of
vertices:");

    scanf("%d",&n);
    //initialise G[] with a null
```

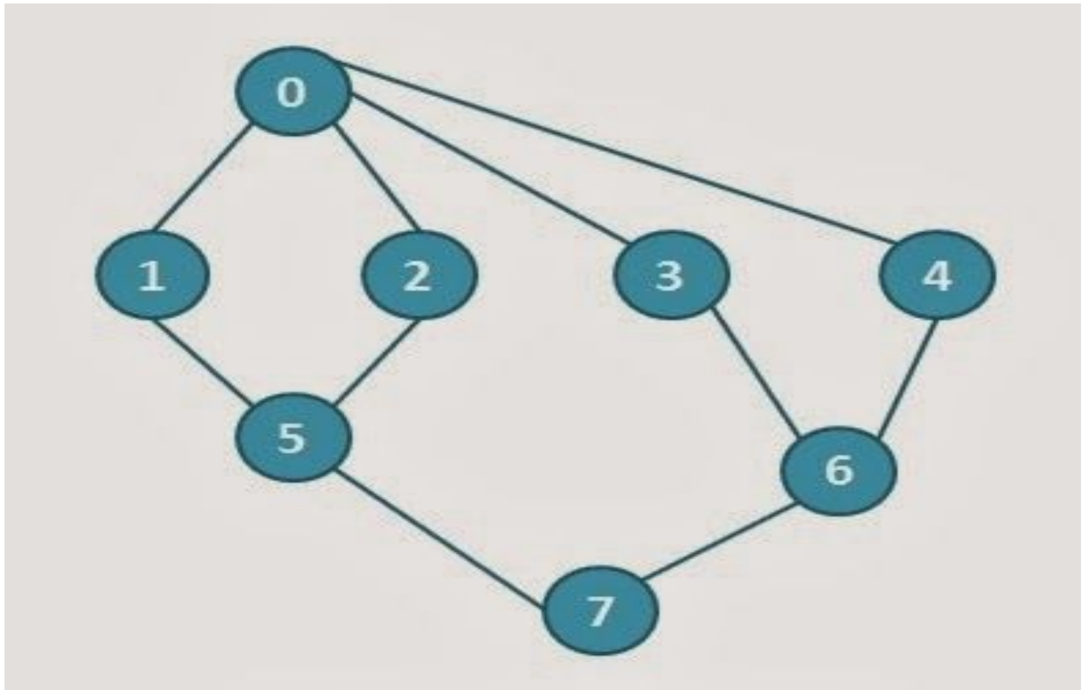
```
  for(i=0;i<n;i++)
   {
      G[i]=NULL;
      //read edges and insert them
in G[]

      printf("Enter number of
edges:");
         scanf("%d",&no_of_edges);
         for(i=0;i<no_of_edges;i++)
      {
          printf("Enter an
edge(u,v):");
          scanf("%d%d",&vi,&vj);
          insert(vi,vj);
      }
   }
}
```

Depth First Search (DFS) Program in C [Adjacency List]

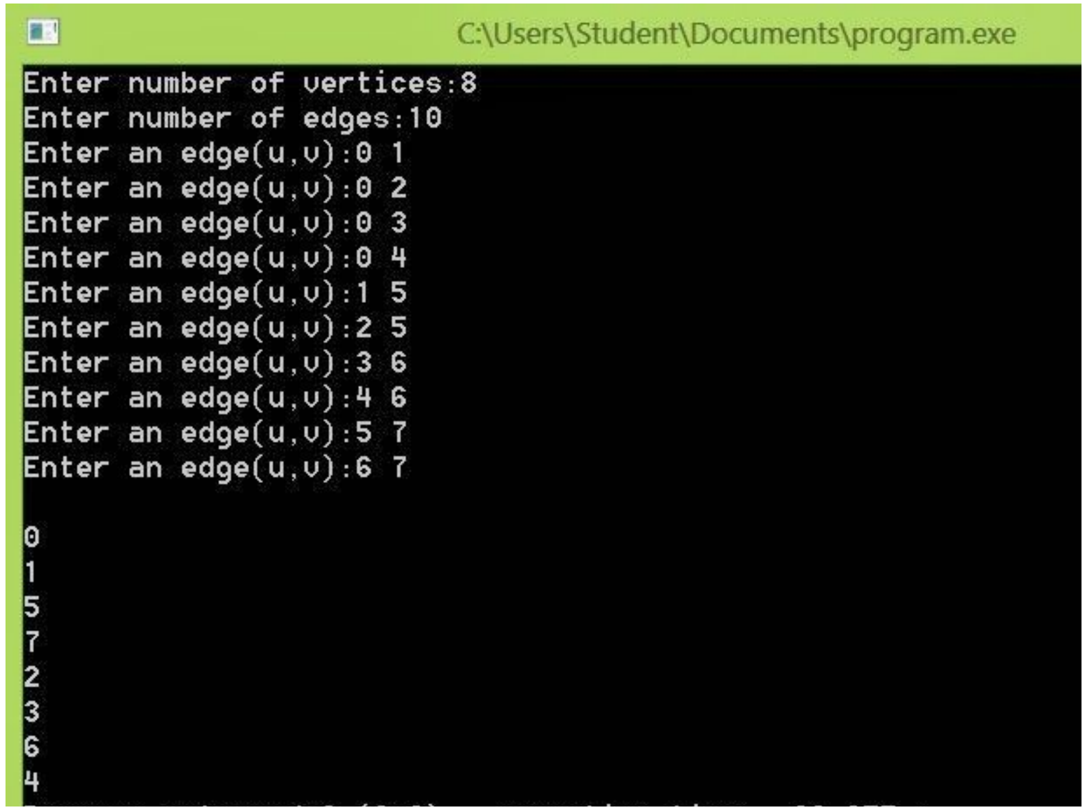
```
void insert(int vi,int vj)
{
   node *p,*q;

   //acquire memory for the new node
   q=(node*)malloc(sizeof(node));
   q->vertex=vj;
   q->next=NULL;
   //insert the node in the linked list number vi
```

```
   if(G[vi]==NULL)
       G[vi]=q;
   else
   {
       //go to end of the linked list
       p=G[vi];

       while(p->next!=NULL)
           p=p->next;
       p->next=q;
   }
}
```

Ouput



Graph used for input



```
Enter number of vertices:8
Enter number of edges:10
Enter an edge(u,v):0 1
Enter an edge(u,v):0 2
Enter an edge(u,v):0 3
Enter an edge(u,v):0 4
Enter an edge(u,v):1 5
Enter an edge(u,v):2 5
Enter an edge(u,v):3 6
Enter an edge(u,v):4 6
Enter an edge(u,v):5 7
Enter an edge(u,v):6 7

0
1
5
7
2
3
6
4
```

The graph shown above is taken as input for the program

Quiz

- Depth First Search is equivalent to which of the traversal in the Binary Trees?
  a) Pre-order Traversal
  b) Post-order Traversal
  c) Level-order Traversal
  d) In-order Traversal

Quiz

- Depth First Search is equivalent to which of the traversal in the Binary Trees?
  a) Pre-order Traversal
  b) Post-order Traversal
  c) Level-order Traversal
  d) In-order Traversal

- Answer: a
  Explanation: In Depth First Search, we explore all the nodes aggressively to one path and then backtrack to the node. Hence, it is equivalent to the pre-order traversal of a Binary Tree.

Quiz

In Depth First Search, how many times a node is visited?
a) Once
b) Twice
c) Equivalent to number of in-degree of the node
d) None of the mentioned

Quiz

In Depth First Search, how many times a node is visited?
a) Once
b) Twice
c) Equivalent to number of indegree of the node
d) None of the mentioned

Answer: c
Explanation: In Depth First Search, we have to see whether the node is visited or not by it's ancestor. If it is visited, we won't let it enter it in the stack.

Analysis of DFS algorithm

How efficient is DFS ?

- Algorithm takes the time proportional to the size of the data structure used for representing the graph in question.
- For adjacent matrix representation, the traversal time is $\Theta(|V|^2)$
- For adjacent list representation, the traversal time is $\Theta(|V|+|E|)$, where $|V|$ and $|E|$ are the number vertices and edges in the graph.

Applications of Depth First Search

Following are the problems that use DFS as a building block.

1) For a weighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) Path Finding
We can specialize the DFS algorithm to find a path between two given vertices u and z.
i) Call DFS(G, u) with u as the start vertex.
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

https://www.geeksforgeeks.org/

Applications of Depth First Search

4) Topological Sorting
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.

5) To test if a graph is bipartite
We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black!

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex

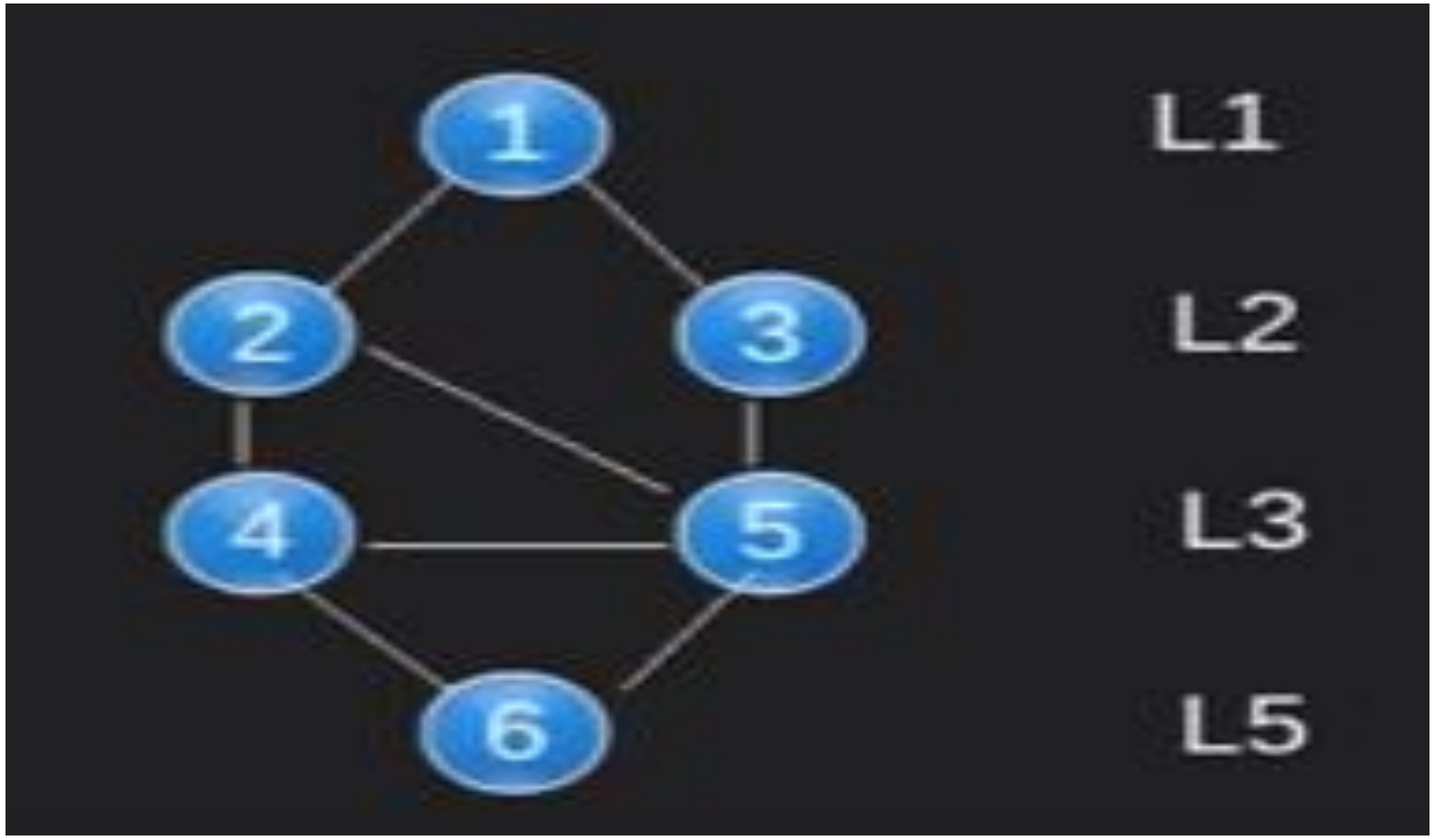
7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

https://www.geeksforgeeks.org/

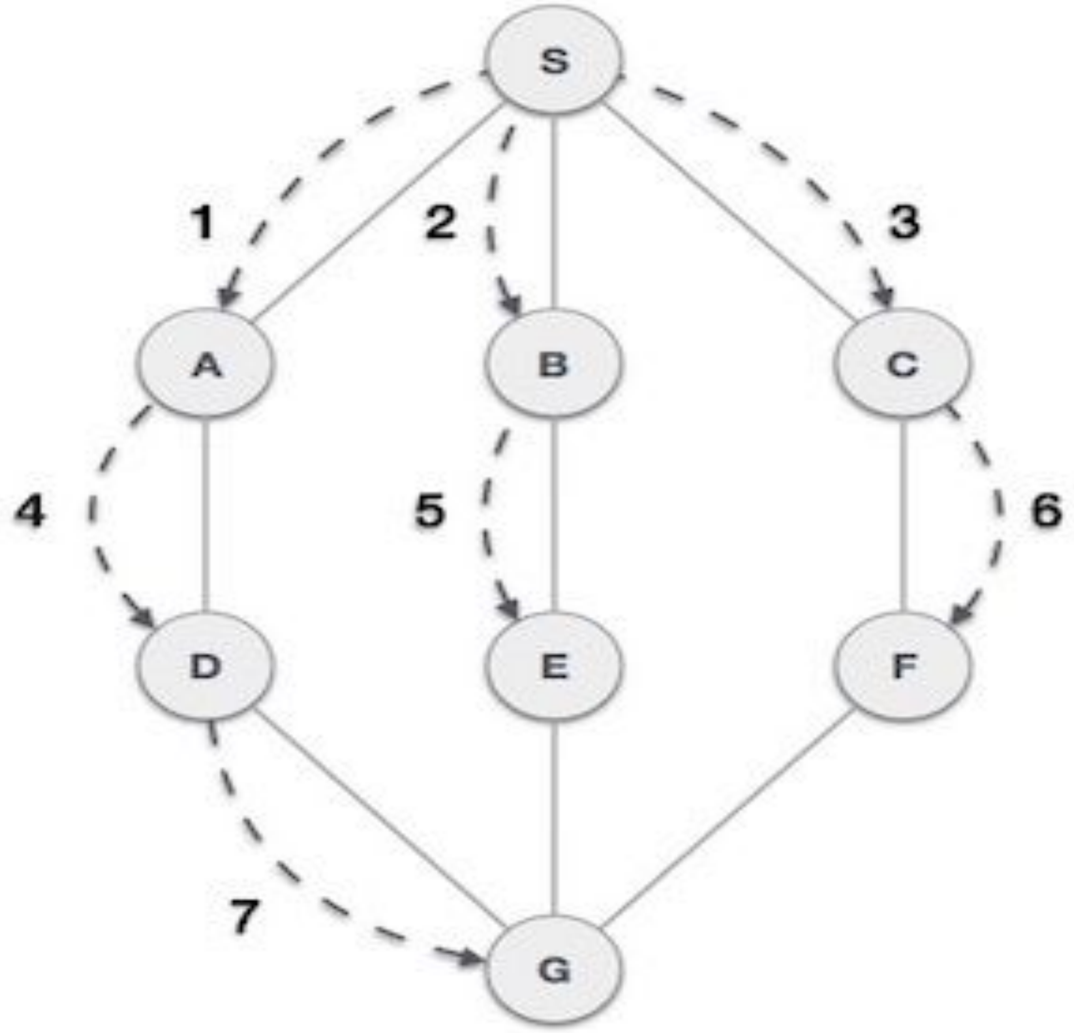
Breadth First Search (BFS)

Breadth First Search (BFS)

- Idea: Traverse graph in levels



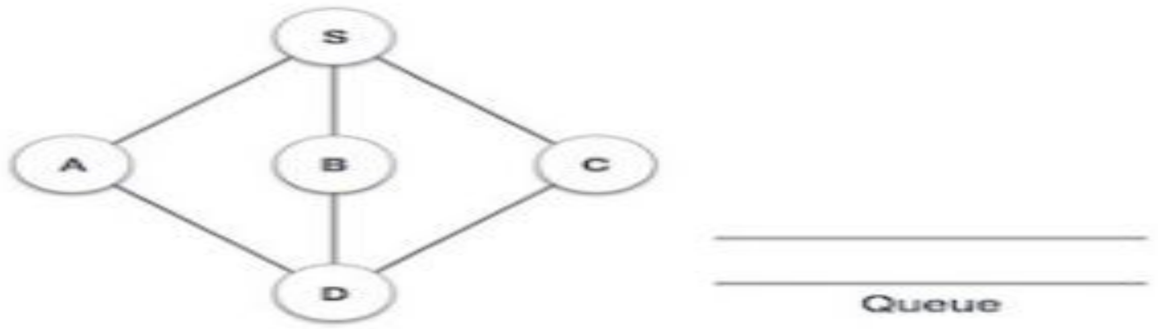
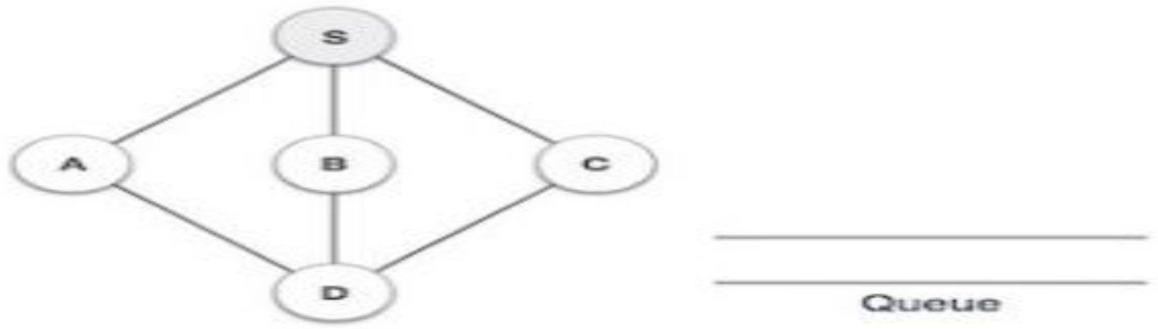
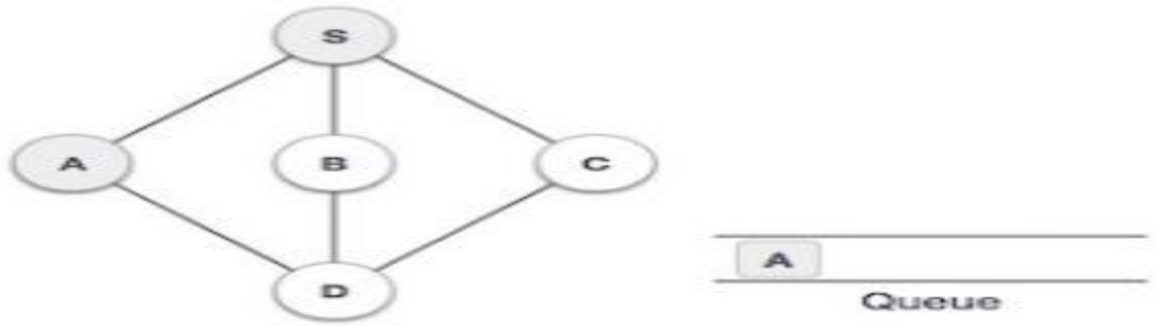
Example: BFS



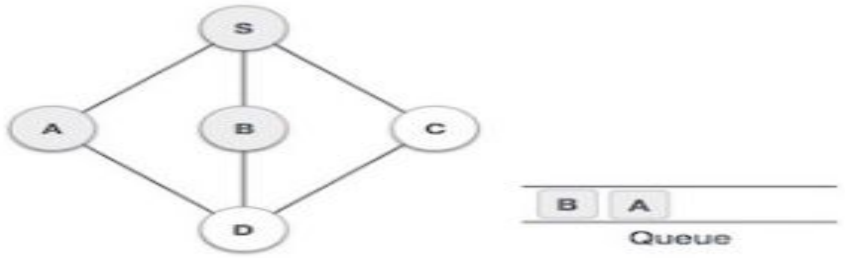
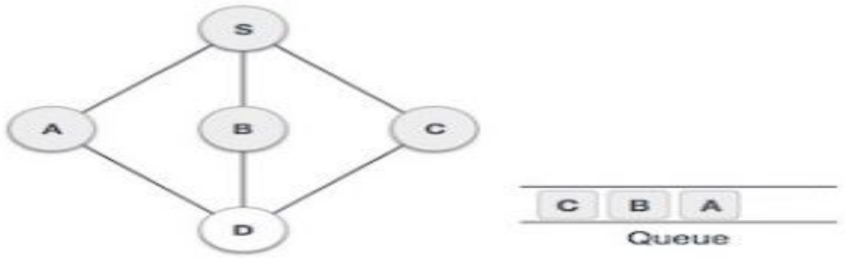
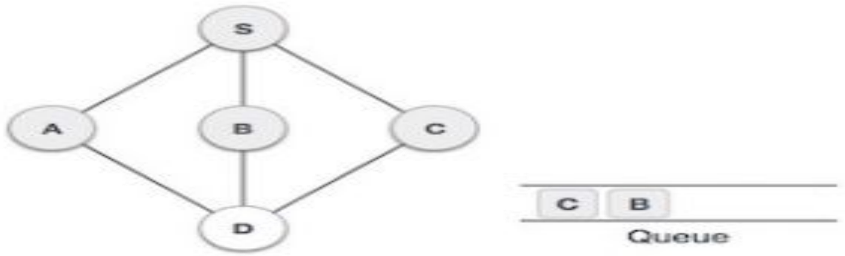
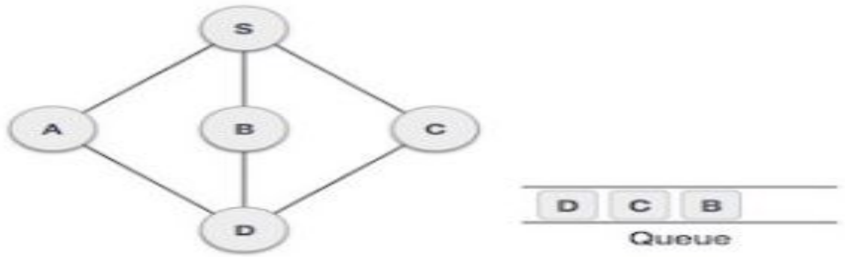
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.
Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty

Example

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting **S** (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

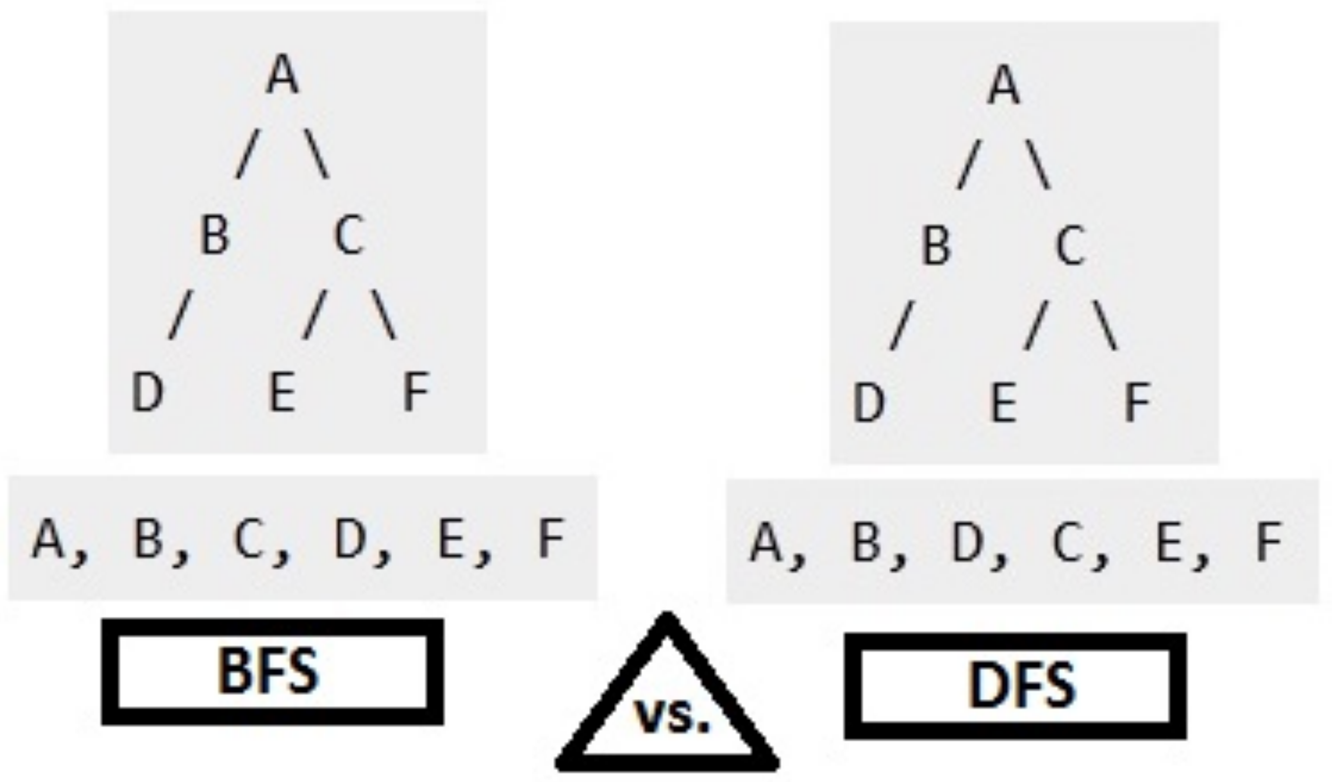
Example

4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6.		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

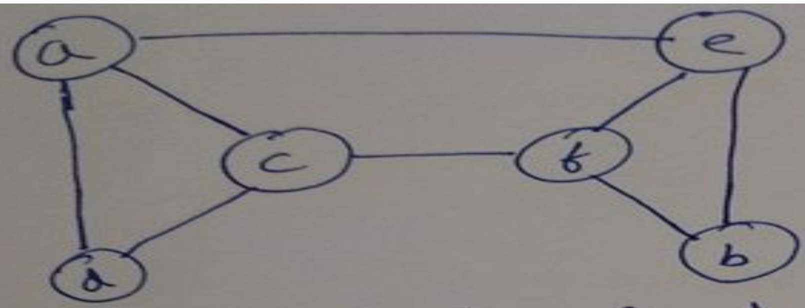
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the algorithm gets terminated.

Example for DFS vs BFS

- Considering A as starting vertex.



Example: BFS



BFS
Starting with a

	a	b	c	d	e	f
Visited status	~~φ~~	~~φ~~	~~φ~~	~~φ~~	~~φ~~	~~φ~~
	1	1	1	1	1	1

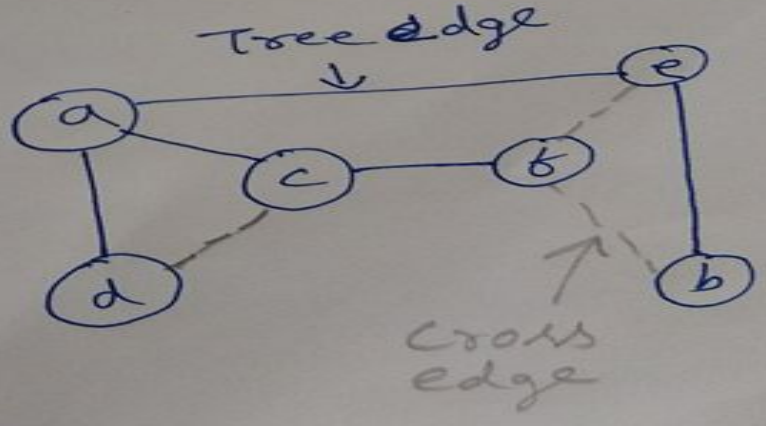
The order in which nodes are visited
Output: a . c d e f

Queue

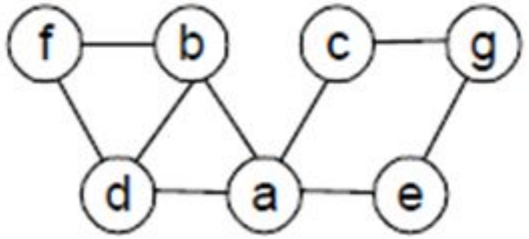
~~a~~	~~c~~	~~d~~	~~e~~	~~f~~	~~b~~

Adjacent vertices

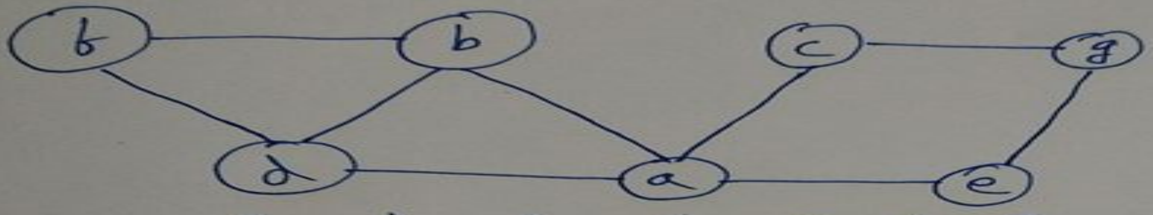
a : ● c d e
c : ~~a~~ ~~d~~ ● f
d : ~~a~~ ~~c~~
e : ~~a~~ b ~~f~~
f : ~~b~~ ~~c~~ ~~e~~
b : ~~e~~ ~~f~~



Example: BFS



Start with a



Visited status

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

The order in which the nodes are visited

Output: a b c d e f g

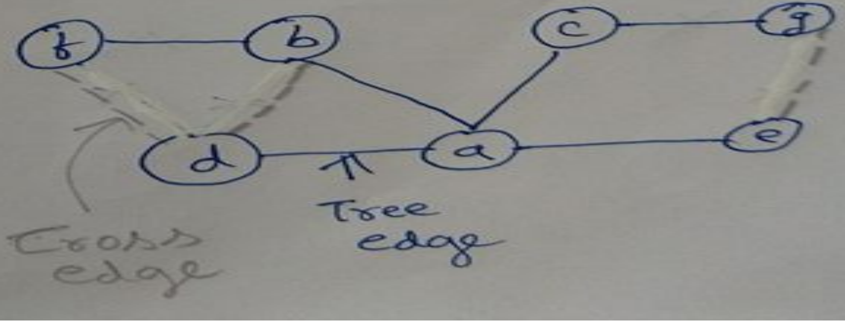
Queue

a	~~b~~	~~c~~	~~d~~	~~e~~	~~f~~	~~g~~

Adjacent vertex

a : b c d e
b : ~~a~~ ~~d~~ f
c : ~~a~~ ~~e~~ g
d : ~~a~~ ~~b~~ ~~f~~
e : ~~a~~ ~~g~~
f : ~~b~~ ~~d~~
g : ~~c~~ ~~e~~

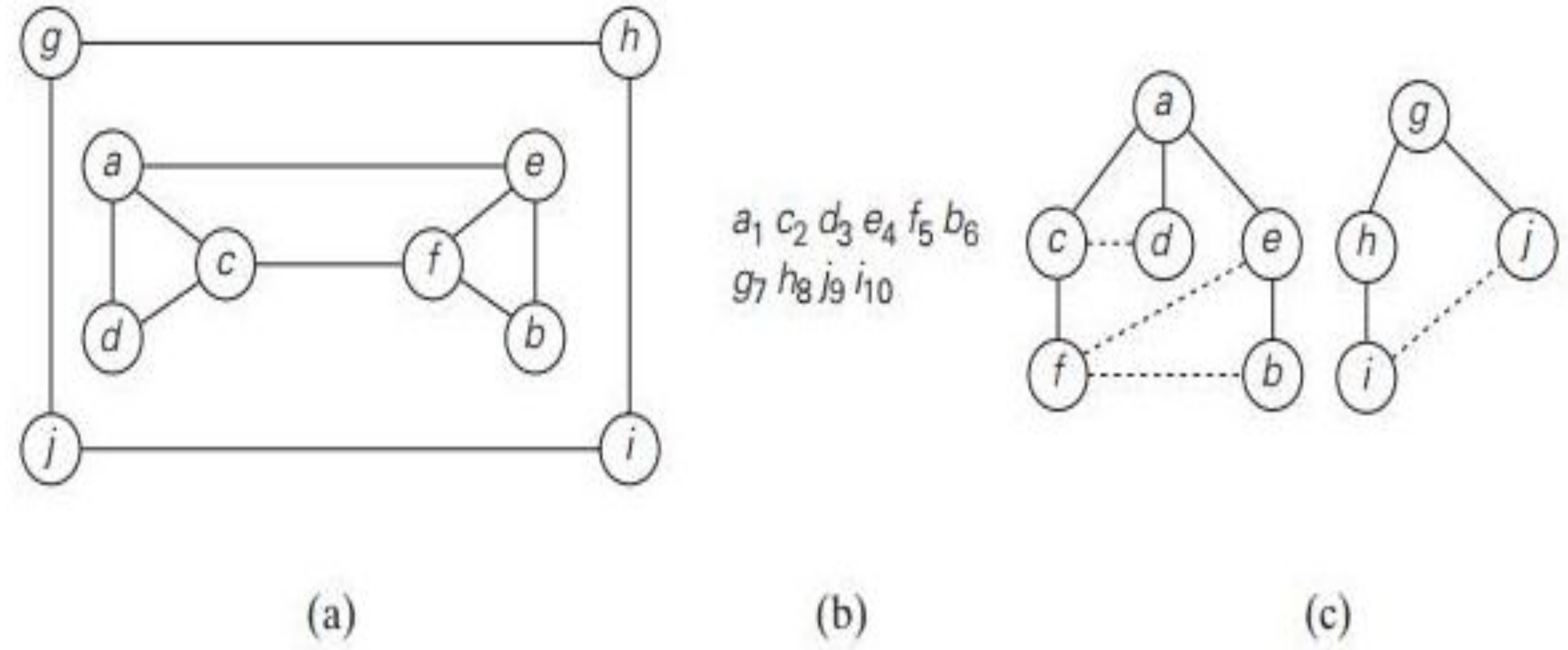
BFS Tree



BFS Algorithm

```
n ← number of nodes
//Initialize visited[ ] to false (0)
for(i=0;i<n;i++)
   visited[i] = 0;

void BFS(K) //BFS starting from vertex K
{
Label K as visited
Initialize Queue  with the vertex K in it
while(Queue is not empty){
Remove front element W from Queue  //Dequeue
For each vertex V adjacent to W {
     if V is not visited{
     add V to the queue                    //Enqueue
     Label V as visited
     Output V
    }
  }
}
}
```

Example



g
h
a
e
c
f
d
b
j
i
a_1 c_2 d_3 e_4 f_5 b_6
g_7 h_8 j_9 i_{10}
a
g
c
d
e
h
j
f
b
i
(a)
(b)
(c)

Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

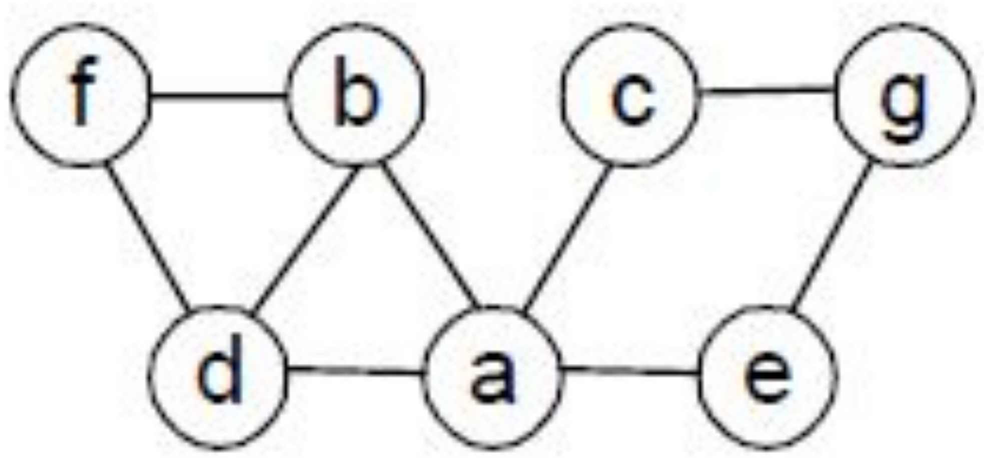
Breadth-first search forest

The traversal’s starting vertex serves as the root of the first tree in such a forest.

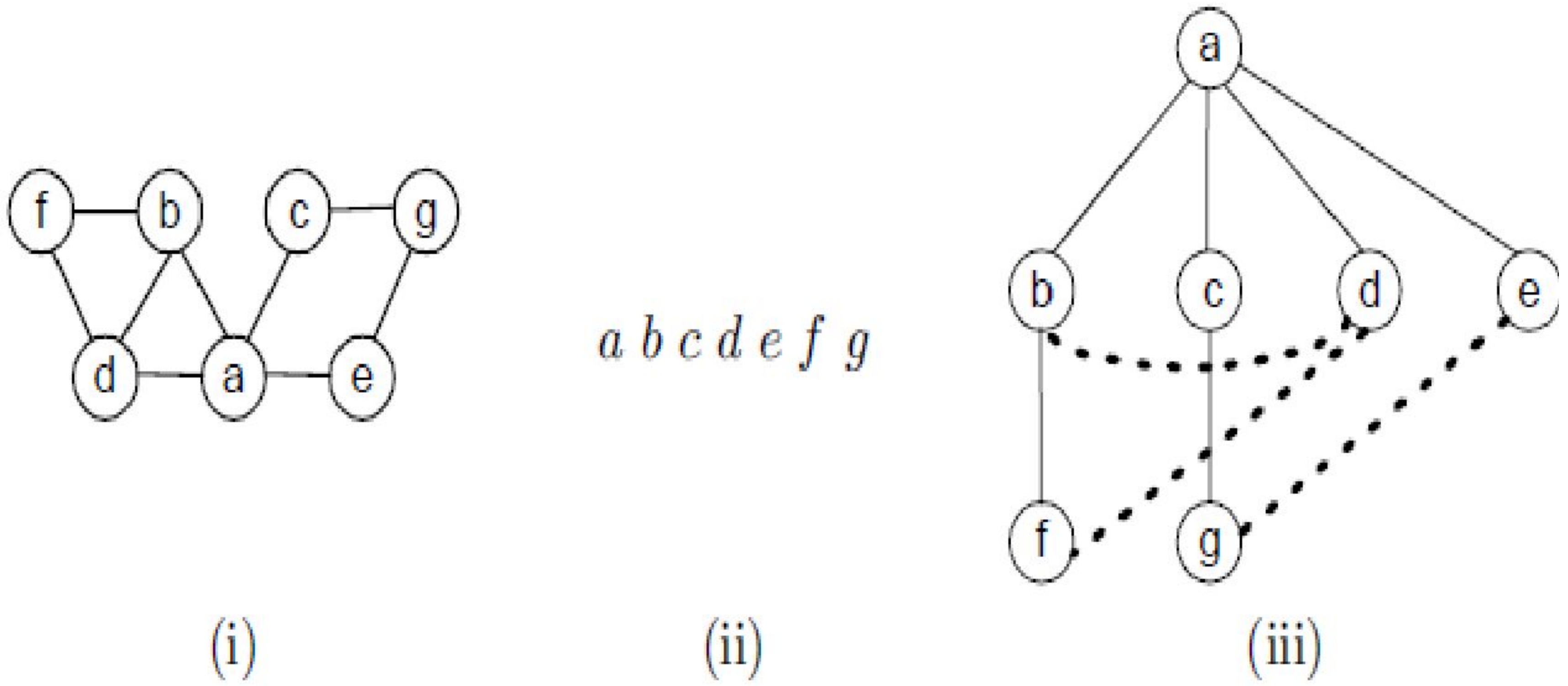
- Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***.
- If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***.

Question

Traverse the graph given below by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex a and resolve ties by the vertex alphabetical order.



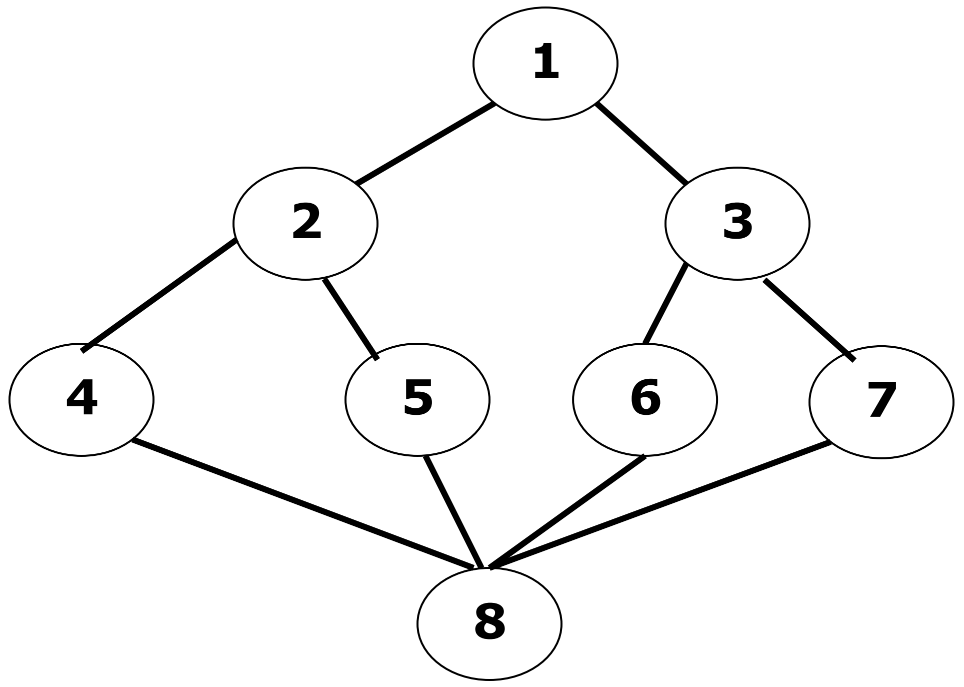
Answer



(i) the graph; (ii) the traversal's queue; (iii) the tree (the tree and cross edges are shown with solid and dotted lines, respectively).

Homework Problem

Traverse the graph given below by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex 1 and resolve ties by the vertex ascending order numbering.



Question

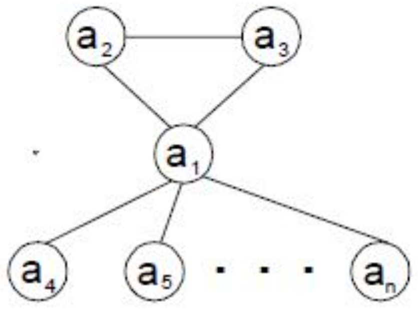
a. Explain how one can check a graph’s acyclicity by using breadth-first search.

b. Does either of the two traversals–DFS or BFS–always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.

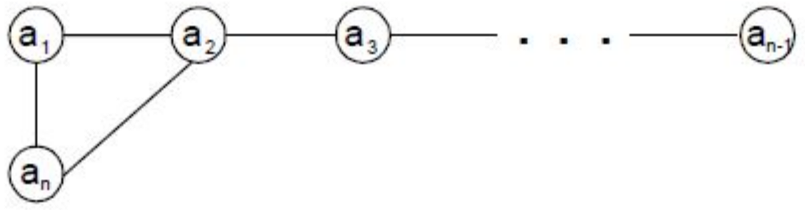
Answer

a. A graph has a cycle if and only if its BFS forest has a cross edge.

b. Both traversals, DFS and BFS, can be used for checking a graph's acyclicity. For some graphs, a DFS traversal discovers a back edge in its DFS forest sooner than a BFS traversal discovers a cross edge (see example (i) below); for others the exactly opposite is the case (see example (ii) below).



(i)



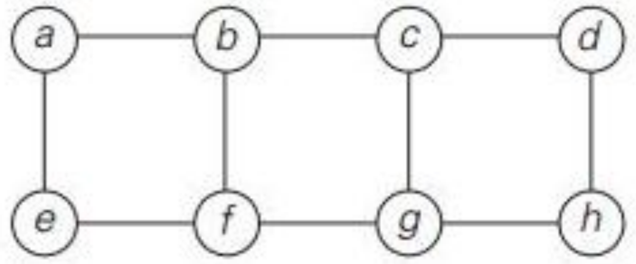
(ii)

Question

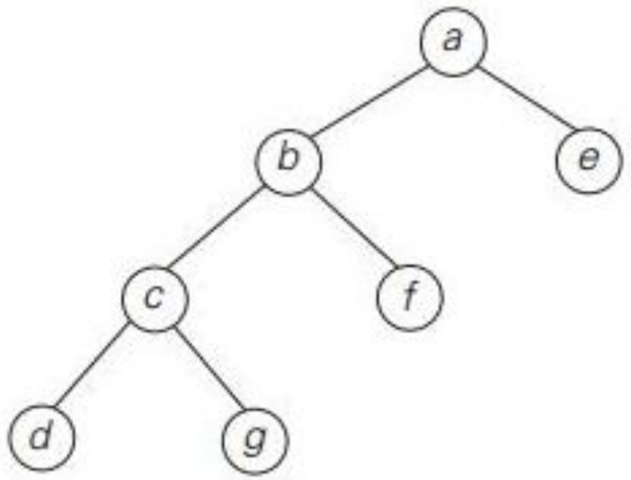
Explain how one can identify connected components of a graph by using
a. depth-first search.
b. breadth-first search.

Answer:
Start a DFS (or BFS) traversal at an arbitrary vertex and mark the visited vertices with 1. By the time the traversal's stack (queue) becomes empty, all the vertices in the same connected component as the starting vertex, and only they, will have been marked with 1. If there are unvisited vertices left, restart the traversal at one of them and mark all the vertices being visited with 2, and so on until no unvisited vertices are left.

Illustration of BFS for finding minimum-edge path



(a)



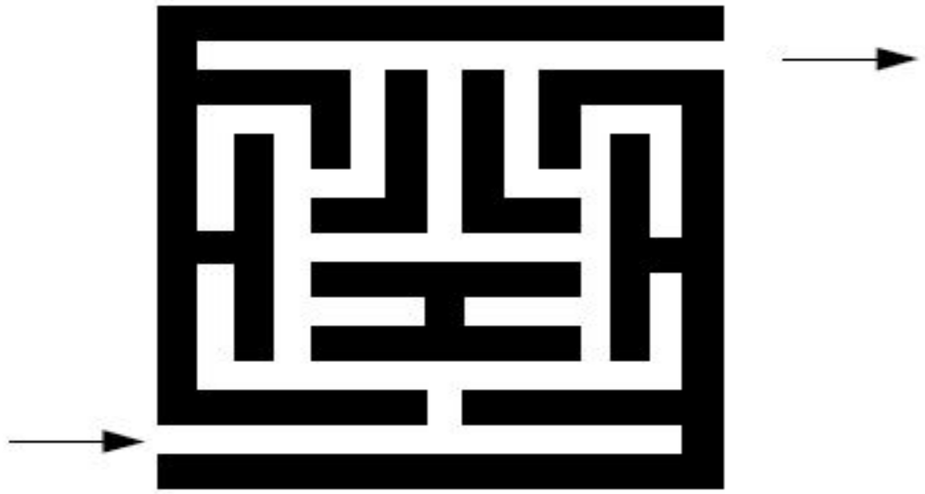
(b)

Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

Question

One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

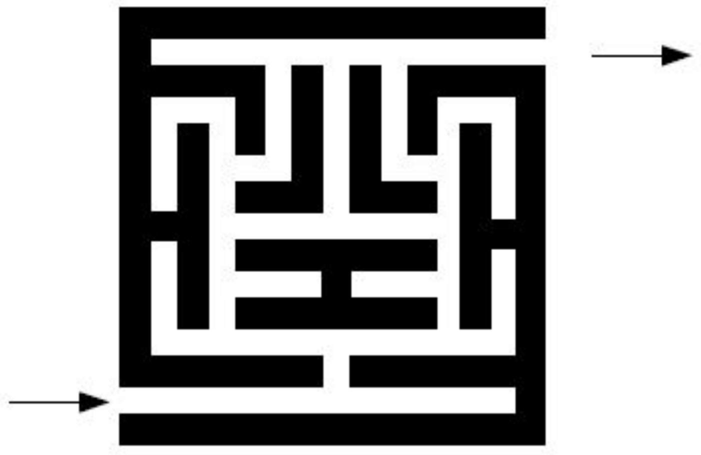
a. Construct such a graph for the following maze.

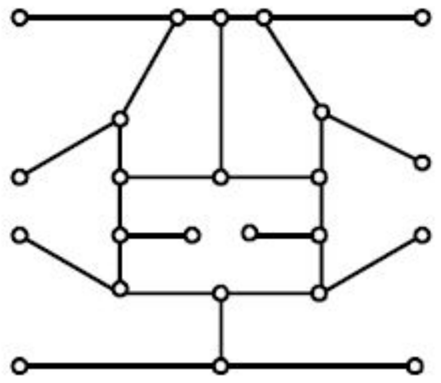


b. Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

Answer

a. Here is the maze and a graph representing it:





b. DFS is much more convenient for going through a maze than BFS. When DFS moves to a next vertex, it is connected to a current vertex by an edge (i.e., “close nearby” in the physical maze), which is not generally the case for BFS. In fact, DFS can be considered a generalization of an ancient right-hand rule for maze traversal: go through the maze in such a way so that your right hand is always touching a wall.

Breadth First Search (BFS) Program in C

```
#include<stdio.h>
#include<stdlib.h>
 #define MAX 100
 #define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);
int queue[MAX], front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
int main()
{
   create_graph();
   BF_Traversal();
   return 0;
}

void BF_Traversal()
{
   int v;

   for(v=0; v<n; v++)
      state[v] = initial;

   printf("Enter Start Vertex for BFS: \n");
   scanf("%d", &v);
   BFS(v);
}
```

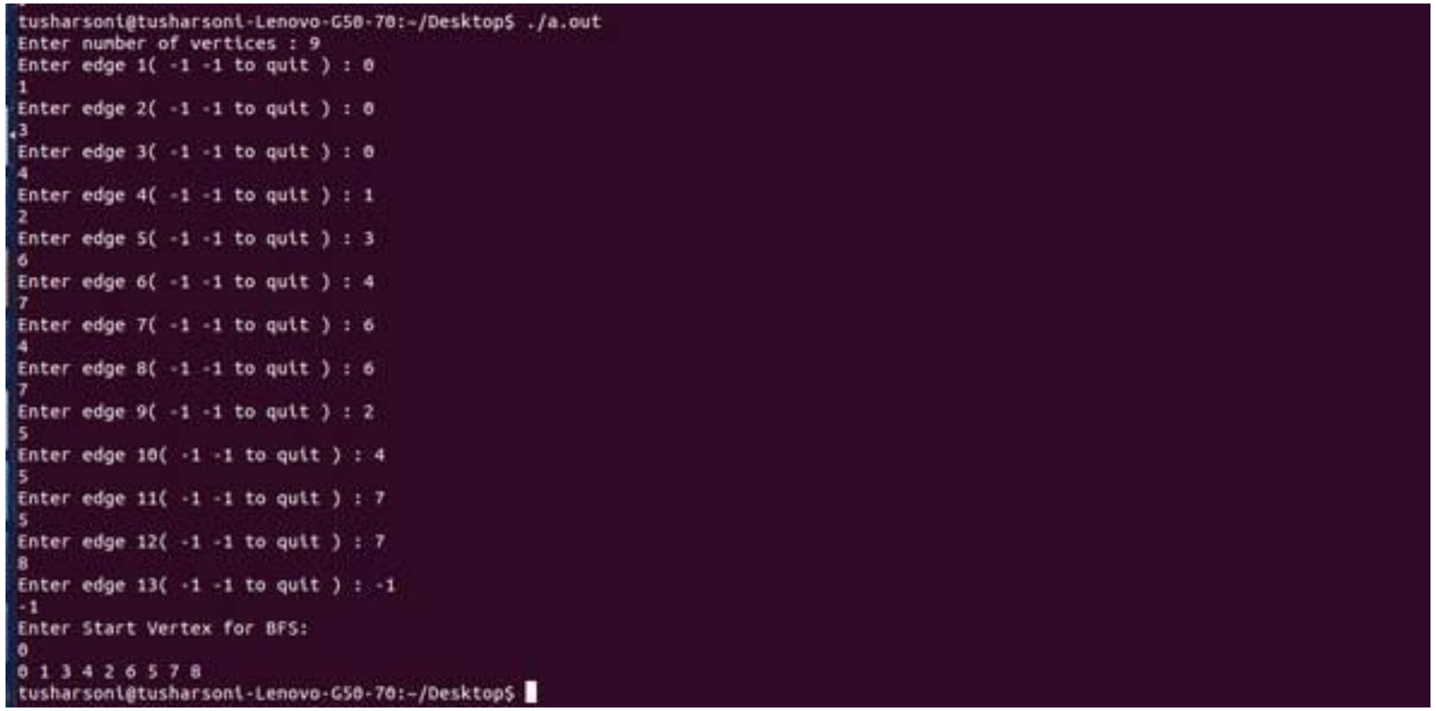
Breadth First Search (BFS) Program in C

```
void BFS(int v)
{   int i;

   insert_queue(v);
   state[v] = waiting;

   while(!isEmpty_queue())
   {
      v = delete_queue( );
      printf("%d ",v);
      state[v] = visited;

      for(i=0; i<n; i++)
      {
         if(adj[v][i] == 1 && state[i] == initial)
         {
            insert_queue(i);
            state[i] = waiting;
         }
      }
   }
   printf("\n");
}
 void insert_queue(int vertex)
{
   if(rear == MAX-1)
      printf("Queue Overflow\n");
   else
   {
      if(front == -1)
         front = 0;
      rear = rear+1;
      queue[rear] = vertex ;
   }
}
```

Breadth First Search (BFS) Program in C

```
int isEmpty_queue()
{
   if(front == -1 || front > rear)
      return 1;
   else
      return 0;
}

int delete_queue()
{
   int delete_item;
   if(front == -1 || front > rear)
   {
      printf("Queue Underflow\n");
      exit(1);
   }

   delete_item = queue[front];
   front = front+1;
   return delete_item;
}

void create_graph()
{
   int count,max_edge,origin,destin;

   printf("Enter number of vertices : ");
   scanf("%d",&n);
   max_edge = n*(n-1);

   for(count=1; count<=max_edge; count++)
   {
      printf("Enter edge %d( -1 -1 to quit ) : ",count);
      scanf("%d %d",&origin,&destin);

      if((origin == -1) && (destin == -1))
         break;

      if(origin>=n || destin>=n || origin<0 || destin<0)
      {
         printf("Invalid edge!\n");
         count--;
      }
      else
      {
         adj[origin][destin] = 1;
      }
   }
}
```

Output



```
tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$ ./a.out
Enter number of vertices : 9
Enter edge 1( -1 -1 to quit ) : 0
1
Enter edge 2( -1 -1 to quit ) : 0
3
Enter edge 3( -1 -1 to quit ) : 0
4
Enter edge 4( -1 -1 to quit ) : 1
2
Enter edge 5( -1 -1 to quit ) : 3
6
Enter edge 6( -1 -1 to quit ) : 4
7
Enter edge 7( -1 -1 to quit ) : 6
4
Enter edge 8( -1 -1 to quit ) : 6
7
Enter edge 9( -1 -1 to quit ) : 2
5
Enter edge 10( -1 -1 to quit ) : 4
5
Enter edge 11( -1 -1 to quit ) : 7
5
Enter edge 12( -1 -1 to quit ) : 7
8
Enter edge 13( -1 -1 to quit ) : -1
-1
Enter Start Vertex for BFS:
0
0 1 3 4 2 6 5 7 8
tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$
```

Analysis of BFS algorithm

How efficient the BFS ?

- BFS efficiency is same as DFS
- Algorithm takes the time proportional to the size of the data structure used for representing the graph in question.
- For adjacent matrix representation, the traversal time is $\Theta(|V|^2)$
- For adjacent list representation, the traversal time is $\Theta(|V|+|E|)$, where $|V|$ and $|E|$ are the number vertices and edges in the graph.

Difference between BFS and DFS

S. No.	Breadth First Search (BFS)	Depth First Search (DFS)
1.	BFS visit nodes **level by level** in Graph.	DFS visit nodes of graph **depth wise**. It visits nodes until reach a leaf or a node which doesn't have non-visited nodes.
2.	A node is fully explored before any other can begin.	Exploration of a node is suspended as soon as another unexplored is found.
3.	Uses Queue data structure to store Un-explored nodes.	Uses Stack data structure to store Un-explored nodes.
4.	BFS is slower and require more memory.	DFS is faster and require less memory.
5.	**Some Applications:** •Finding all connected components in a graph. •Finding the shortest path between two nodes. •Finding all nodes within one connected component. •Testing a graph for bipartiteness.	**Some Applications:** •Topological Sorting. •Finding connected components. •Solving puzzles such as maze. •Finding strongly connected components. •Finding articulation points (cut vertices) of the graph

Lab Program 4

Write program to do the following:

a. Print all the nodes reachable from a given starting node in a digraph using BFS method.

b. Check whether a given graph is connected or not using DFS method.

Thanks for Listening

References

- https://www.geeksforgeeks.org/applications-of-depth-first-search/
- Time Complexity: https://www.youtube.com/watch?v=9TlHvipP5yA