

ANALYSIS AND DESIGN OF ALGORITHMS

UNIT-II

CHAPTER 5:

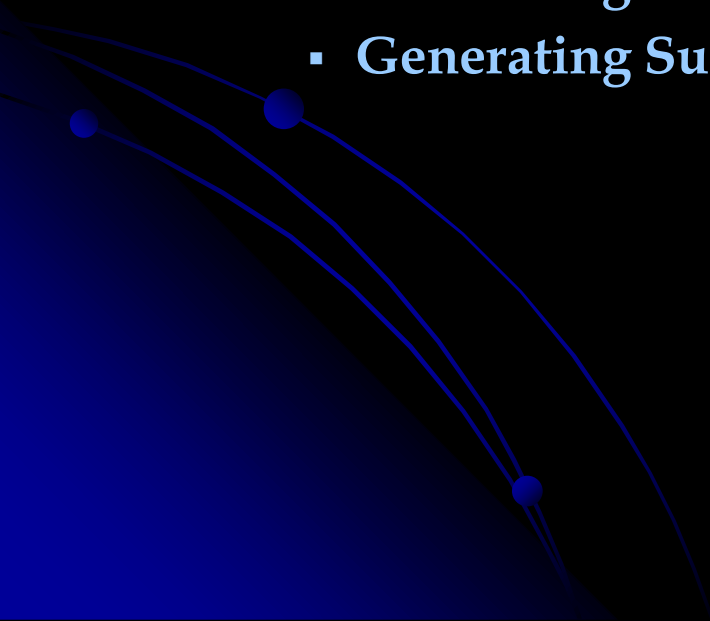
DECREASE-AND-CONQUER



OUTLINE :

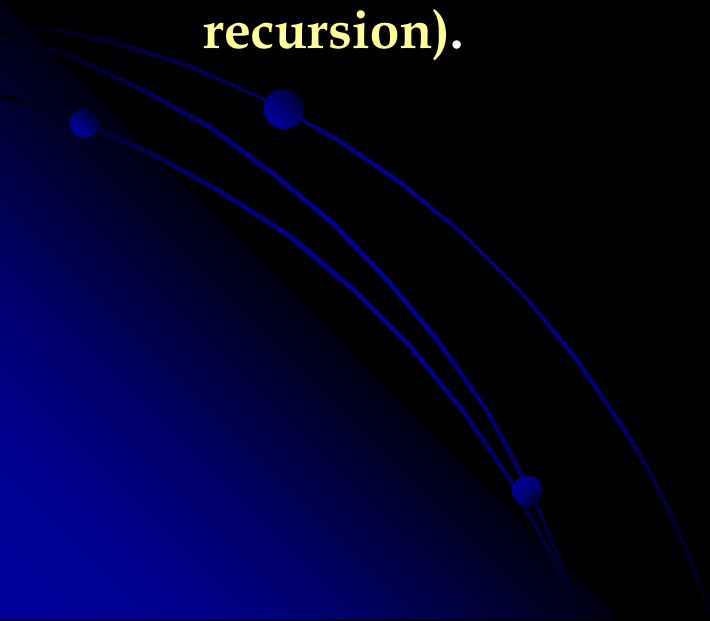
➤ Decrease-and-Conquer

- Insertion Sort
- Depth-First Search
- Breadth-First Search
- Topological Sorting
- Algorithms for Generating Combinatorial Objects
 - Generating Permutations
 - Generating Subsets



Decrease-and-Conquer

- The decrease-and-Conquer technique is based on **exploiting the relationship** between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- Once such a relationship is established, it can be exploited either **top down (recursively)** or **bottom up (without a recursion)**.



Decrease-and-Conquer

There are three major variations of decrease-and-conquer:

➤ Decrease by a constant:

- Insertion sort
- Graph algorithms:
 - DFS
 - BFS
 - Topological sorting
- Algorithms for generating permutations, subsets

➤ Decrease by a constant factor

- Binary search

➤ Variable-size decrease

- Euclid's algorithm

- In the **decrease-by-a-constant variation**, the size of an instance is reduced by the same constant (typically, this constant is equal to one) on each iteration of the algorithm.

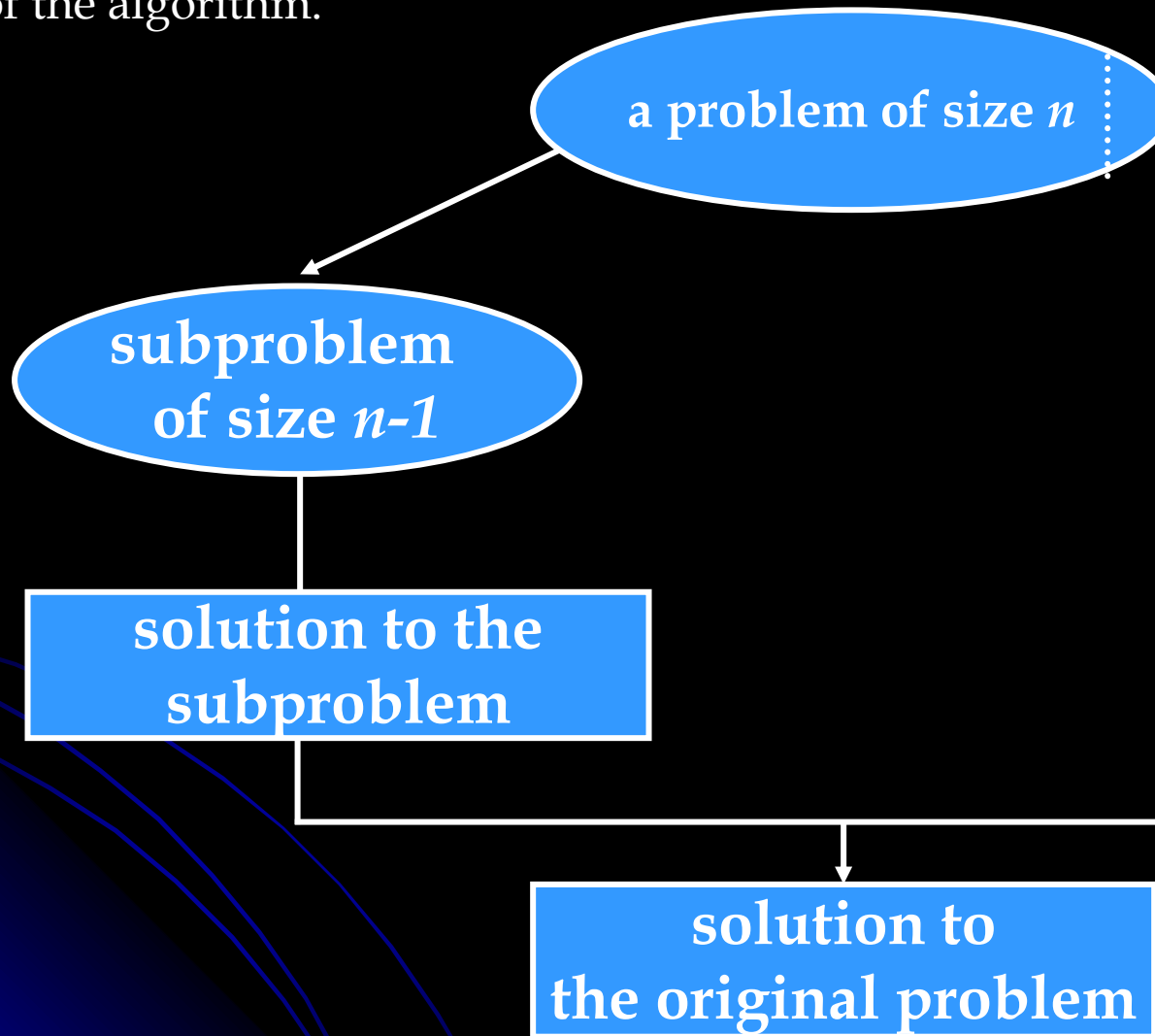


Figure : Decrease (by one) - and - conquer technique.

Decrease-by-a-Constant

➤ For example, **consider the exponentiation problem of computing a^n** for positive integer exponents:

- The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the formula:

$$a^n = a^{n-1} \cdot a$$

- So, the function $f(n) = a^n$ can be computed “**top down**” by using its recursive definition:

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

- Function $f(n) = a^n$ can be computed “**bottom up**” by multiplying a by itself $n-1$ times.

- In the **decrease-by-a-constant factor variation**, the size of a problem instance is reduced by **the same constant factor** on each iteration of the algorithm. In most applications, this constant factor is equal to two.

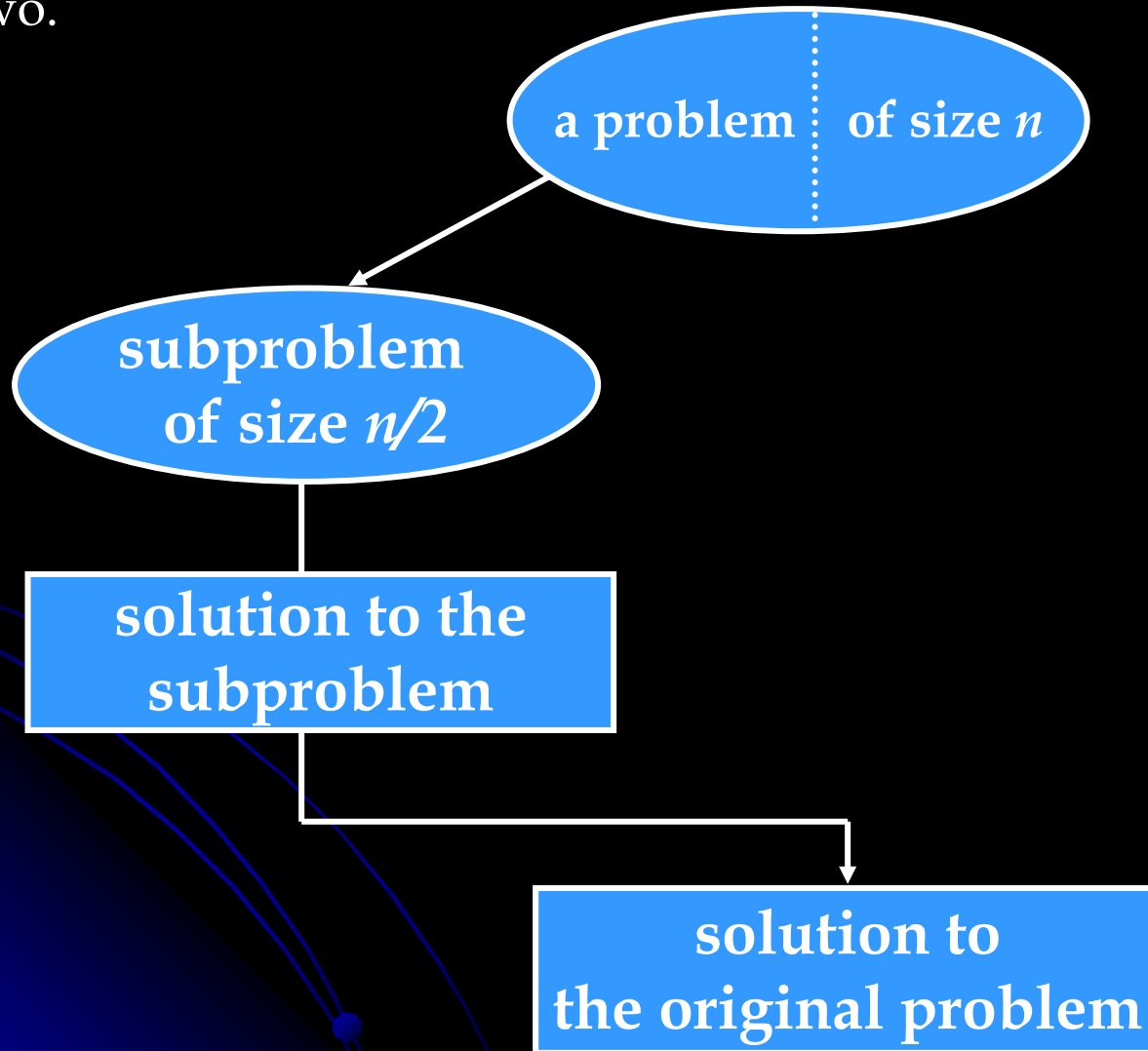


Figure : Decrease (by half) - and - conquer technique.

Decrease – by – a – constant – factor:

➤ For example, **consider the exponentiation problem of computing a^n** for positive integer exponents:

- If the instance of size n is to compute a^n , the instance of half its size will be to compute $a^{n/2}$, with obvious relationship between the two:
 $a^n = (a^{n/2})^2$
- If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a .
- To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \\ a & \text{if } n = 1 \end{cases}$$

- If we compute a^n recursively according to above formula and measure the algorithm's efficiency by number of multiplications, then algorithm is expected to be in $O(\log n)$ because, on each iteration, the size is reduced by at least one half at the expense of no more than two multiplications.

Consider the problem of exponentiation: Compute a^n

➤ Brute Force:

$$a^n = a * a * a * a * \dots * a$$

➤ Divide and conquer:

$$a^n = a^{n/2} * a^{n/2}$$

➤ Decrease by one:

$$a^n = a^{n-1} * a$$

➤ Decrease by constant factor: $a^n = (a^{n/2})^2$ if n is even

$$a^n = (a^{(n-1)/2})^2 * a \text{ if } n \text{ is odd}$$

Variable – Size – Decrease:

- In this variation, **the size reduction pattern varies from one iteration of an algorithm to another.**
- Euclid's algorithm for computing the GCD provides a good example of such a situation:

$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

- Though the arguments on the right-hand side are always smaller than those on the left-hand side, they are smaller neither by a constant nor by a constant factor.

Insertion Sort

- **Decrease – by – one technique** is applied to sort an array $A[0 \dots n-1]$.
- We assume that the smaller problem of sorting the array $A[0 \dots n-2]$ has already been solved to give us a sorted array of size $n - 1$:

$$A[0] \leq \dots \leq A[n - 2].$$

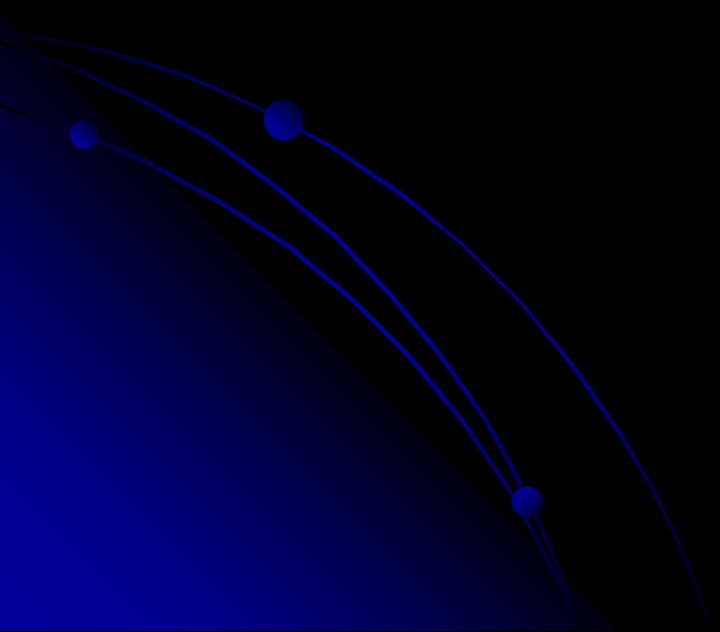
- How can we take the advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n-1]$?
 - All we need is to find an appropriate position for $A[n - 1]$ among sorted elements and insert it there.

Insertion Sort

- There are **three reasonable alternatives** for finding an appropriate position for $A[n - 1]$ among sorted elements:
 - First, we can **scan the sorted subarray from left to right** until the first element greater than or equal to $A[n - 1]$ is encountered and then insert $A[n - 1]$ right before that element.
 - Second, we can **scan the sorted subarray from right to left** until the first element smaller than or equal to $A[n - 1]$ is encountered and then insert $A[n - 1]$ right after that element.
 - This is implemented in practice because it is better for sorted or almost-sorted arrays. The resulting algorithm is called **straight insertion sort** or simply **insertion sort**.

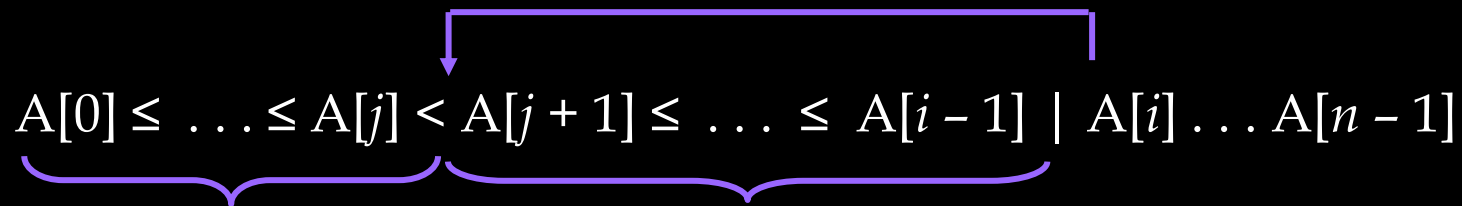
Insertion Sort

- The third alternative is to use **binary search** to find an appropriate position for $A[n - 1]$ in the sorted portion of the array. The resulting algorithm is called **binary insertion sort**.
 - Improves the number of comparisons (worst-case and average-case).
 - Still requires same number of swaps.



Insertion Sort

As shown in below figure, starting with $A[1]$ and ending with $A[n - 1]$, $A[i]$ is inserted in its appropriate place among the first i elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

$$\underbrace{A[0] \leq \dots \leq A[j]}_{\text{Smaller than or equal to } A[i]} < \underbrace{A[j+1] \leq \dots \leq A[i-1]}_{\text{greater than } A[i]} \mid A[i] \dots A[n-1]$$


Smaller than or equal to $A[i]$

greater than $A[i]$

Pseudocode of Insertion Sort

ALGORITHM *InsertionSort*($A[0..n - 1]$)

//Sorts a given array by insertion sort

//Input: An array $A[0..n - 1]$ of n orderable elements

//Output: Array $A[0..n - 1]$ sorted in nondecreasing order

for $i \leftarrow 1$ **to** $n - 1$ **do**

$v \leftarrow A[i]$

$j \leftarrow i - 1$

while $j \geq 0$ **and** $A[j] > v$ **do**

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

Insertion Sort program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void insertion(int*,int);
    int a[10],n,i;

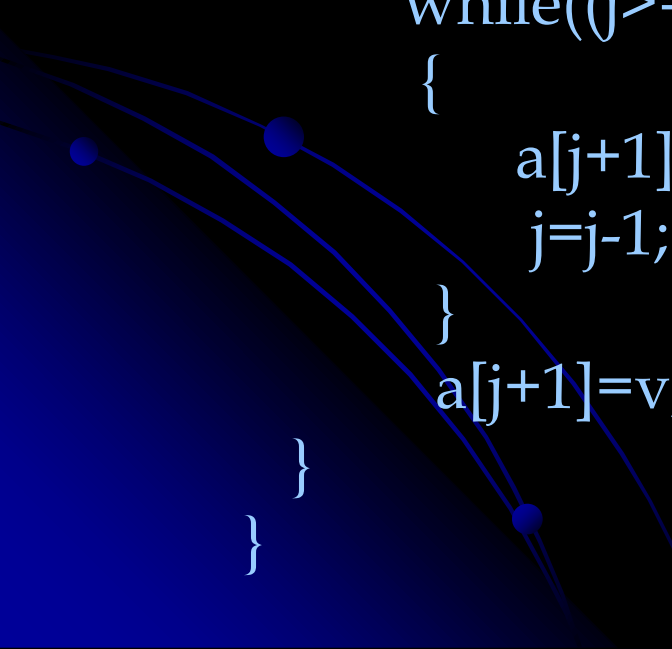
    printf("\nEnter the size of the array\n");
    scanf("%d",&n);

    printf("\nEnter the array elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("\nArray before sorting\n");
    for(i=0;i<n;i++)
        printf("%d\n",a[i]);
    insertion(a,n);
    printf("\nArray after sorting\n");
    for(i=0;i<n;i++)
        printf("%d\n",a[i]);
}
```

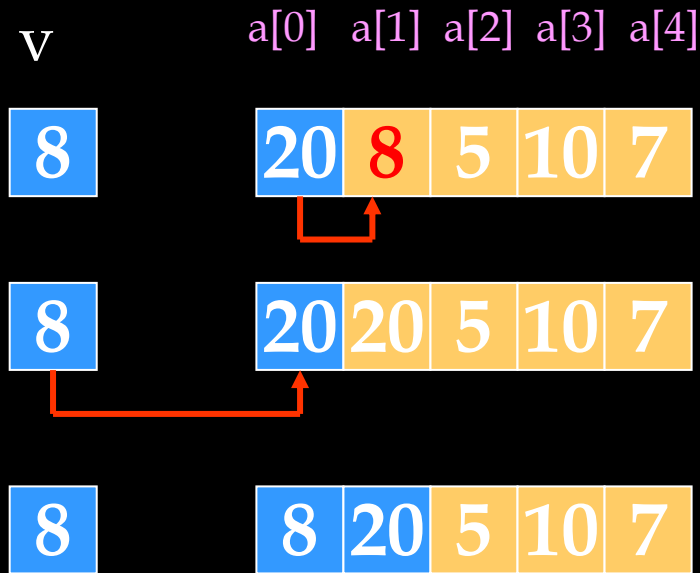

Insertion Sort program:

```
void insertion(int *a,int n)
{
    int i,j,v;
    for(i=1;i<n;i++)
    {
        v=a[i];
        j=i-1;
        while((j>=0) && (a[j] > v))
        {
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=v;
    }
}
```



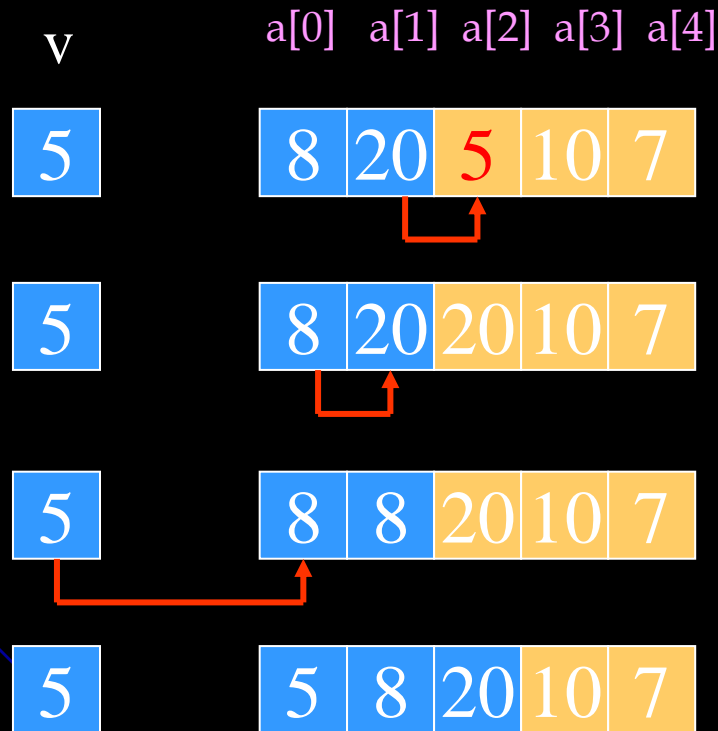
Insertion Sort Example:

Insert Action: $i=1$, first iteration



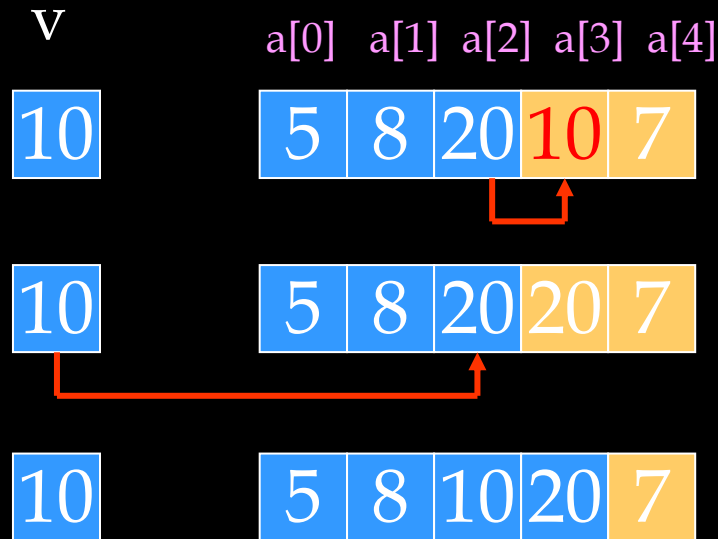
Insertion Sort Example:

Insert Action: $i=2$, second iteration



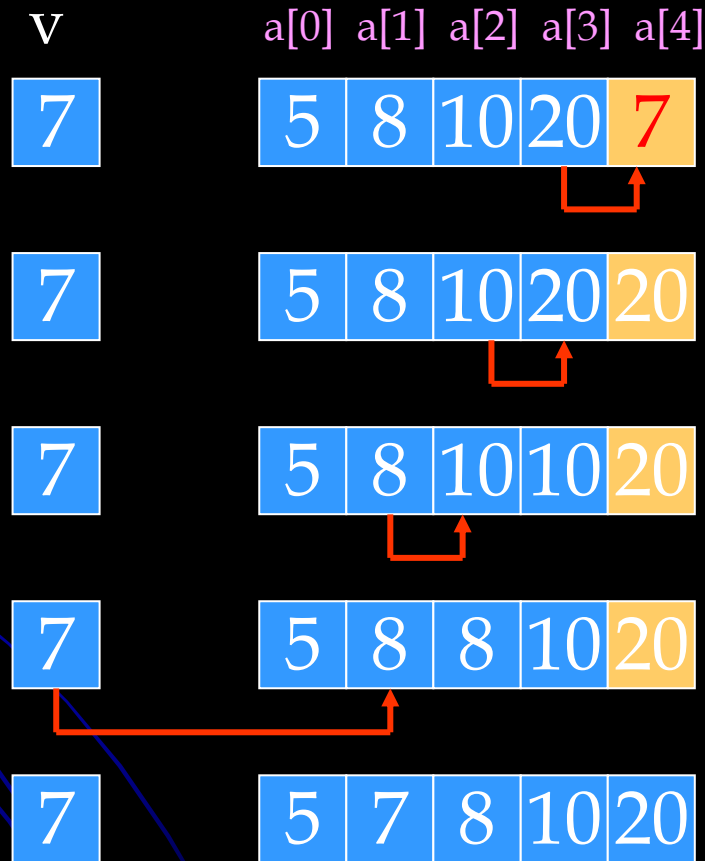
Insertion Sort Example:

Insert Action: $i=3$, third iteration



Insertion Sort Example:

Insert Action: $i=4$, fourth iteration



Sorted ARRAY

Insertion Sort Example 1:

20		8	5	10	7			
8		20		5	10	7		
5		8		20		10	7	
5		8		10		20		7
5		7		8		10		20

Figure: A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Insertion Sort Example 2:

89		45	68	90	29	34	17
45	89		68	90	29	34	17
45	68	89		90	29	34	17
45	68	89	90		29	34	17
29	45	68	89	90		34	17
29	34	45	68	89	90		17
17	29	34	45	68	89	90	

Figure: A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Insertion Sort

Problems:

Apply insertion sort to sort the following list in alphabetical order:

1. I, N, S, E, R, T, I, O, N, S, O, R, T

2. D, E, C, R, E, A, S, E, A, N, D, C, O, N, Q, U, E, R



Insertion Sort

Analysis:

- The **number of key comparisons** in this algorithm obviously depends on the nature of the input.
- In the **worst case**, $A[j] > v$ is executed the **largest number of times**. i.e., for every $j = i - 1, \dots, 0$.
 - Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), \dots $A[n - 2] > A[n - 1]$ (for $i = n - 1$).
 - The **worst-case input** is an **array of strictly decreasing values**. The no. of key comparisons for such an input is

$$C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = ((n-1)n)/2 \in \Theta(n^2)$$

- Thus, in the worst-case, insertion sort makes exactly the same number of comparisons as selection sort.

Insertion Sort

Analysis:

- In the **Best case**, the comparison $A[j] > v$ is executed **only once** on every iteration of outer loop.
 - It happens if and only if $A[i - 1] \leq A[i]$ for every $i = 1, \dots, n - 1$, i.e., if the input array is **already sorted in ascending order**.

$$C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Insertion Sort

Average-case Analysis:

Consider the already sorted array with 1 element 10.

- The element 12 need to be inserted to this array with single element.
 - No. of times while loop condition is checked = 1
- The element 8 need to be inserted to this array with single element.
 - No. of times while loop condition is checked = 2

Therefore, Average = $(1+2)/2 = 1$.

Insertion Sort

Average-case Analysis:

Consider the already sorted array with 2 elements 10, 12.

- The element 13 need to be inserted to this array with two elements.
 - No. of times while loop condition is checked = 1
- The element 11 need to be inserted to this array with two elements.
 - No. of times while loop condition is checked = 2
- The element 8 need to be inserted to this array with two elements.
 - No. of times while loop condition is checked = 3

Therefore, Average = $(1+2+3)/3 = 2$.

Insertion Sort

Average-case Analysis:

Consider the already sorted array with 3 elements 10, 12, 14.

- The element 15 need to be inserted to this array with three elements.
 - No. of times while loop condition is checked = 1
- The element 13 need to be inserted to this array with three elements.
 - No. of times while loop condition is checked = 2
- The element 11 need to be inserted to this array with three elements.
 - No. of times while loop condition is checked = 3
- The element 8 need to be inserted to this array with three elements.
 - No. of times while loop condition is checked = 4

Therefore, Average = $(1+2+3+4)/4 = 2$.

Insertion Sort

Average-case Analysis:

Similarly, to insert an element X with index i , in the proper position, the total no. of times while loop condition is checked on an average is given by

$$\frac{(1 + 2 + \dots + i)}{i} = \frac{i(i+1)}{2 \cdot i} = \frac{(i+1)}{2}$$

$$T(n) = \sum_{i=1}^{n-1} (i+1)/2 = 1/2 \sum_{i=1}^{n-1} (i+1)$$

$$= 1/2 \sum_{i=1}^{n-1} i + 1/2 \sum_{i=1}^{n-1} 1$$

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

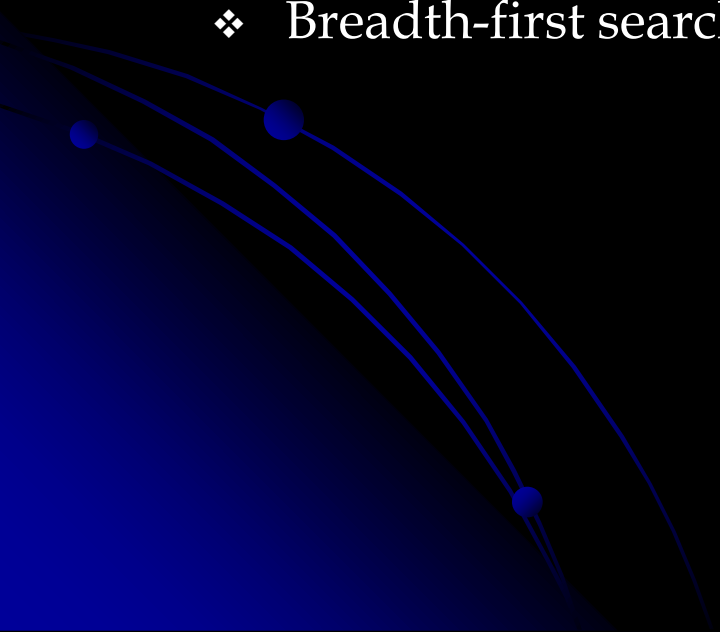
$$= n^2/4 - n/4 + n/2 - 1/2$$

$$= n^2/4 + n/4 - 1/2$$

Therefore, $C_{\text{avg}}(n) \approx n^2/4 \in \Theta(n^2)$

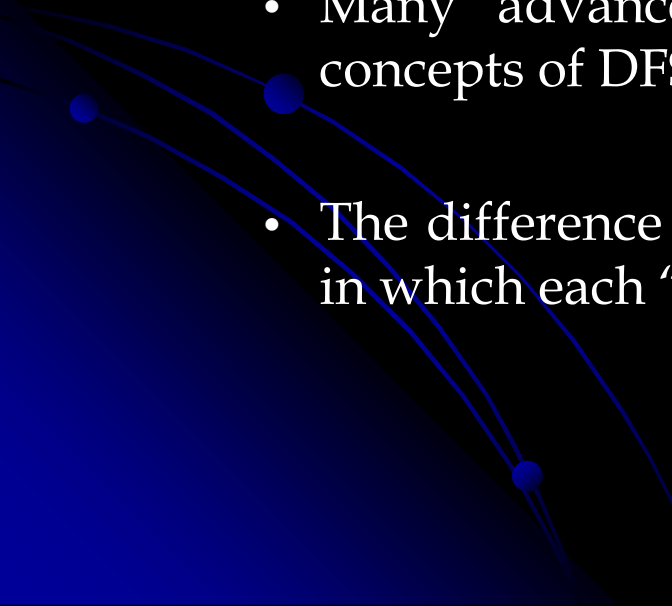
Graph Traversal

- Many graph algorithms require processing vertices or edges of a graph in a systematic fashion.
- **There are two principal Graph traversal algorithms:**
 - ❖ Depth-first search (DFS)
 - ❖ Breadth-first search (BFS)



Graph Traversal

➤ **Depth-First Search (DFS) and Breadth-First Search (BFS):**

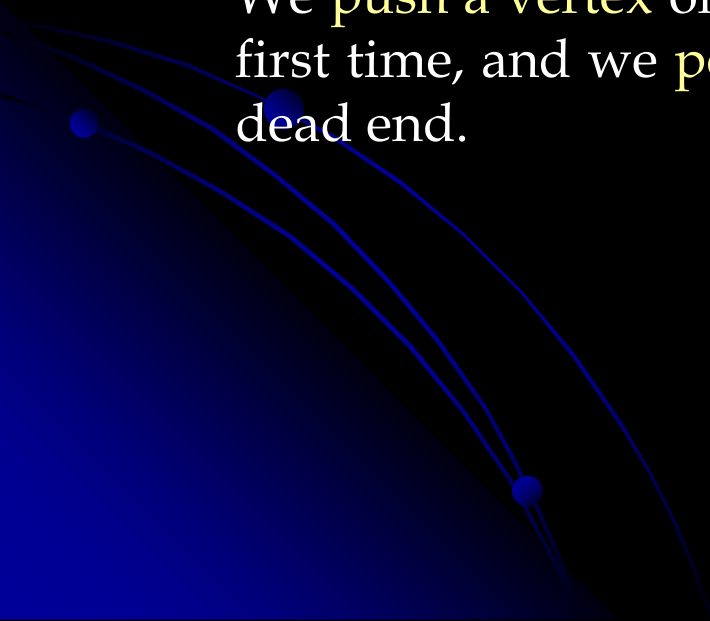
- Two elementary traversal algorithms that provide an efficient way to “visit” each vertex and edge exactly once.
 - Both work on directed or undirected graphs.
 - Many advanced graph algorithms are based on the concepts of DFS or BFS.
 - The difference between the two algorithms is in the order in which each “visit” vertices.
- 

Depth-First Search

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

Depth-First Search

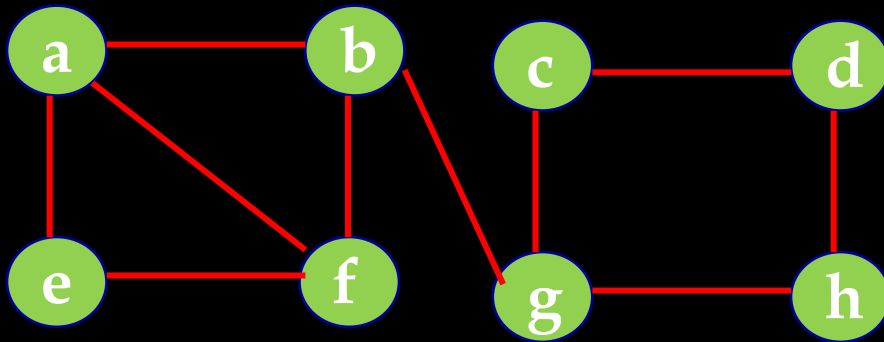
- By this time, all the vertices in the same connected component have been visited.
- If unvisited vertices still remain, the depth-first search must be restarted at any one of them.
- It is convenient to **use a stack** to trace the operation of DFS.
We **push a vertex** onto the stack when the vertex is reached for first time, and we **pop a vertex** off the stack when it becomes a dead end.



Depth-First Search

- It is also very useful to accompany a depth-first search traversal by constructing the so-called **depth-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, 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.

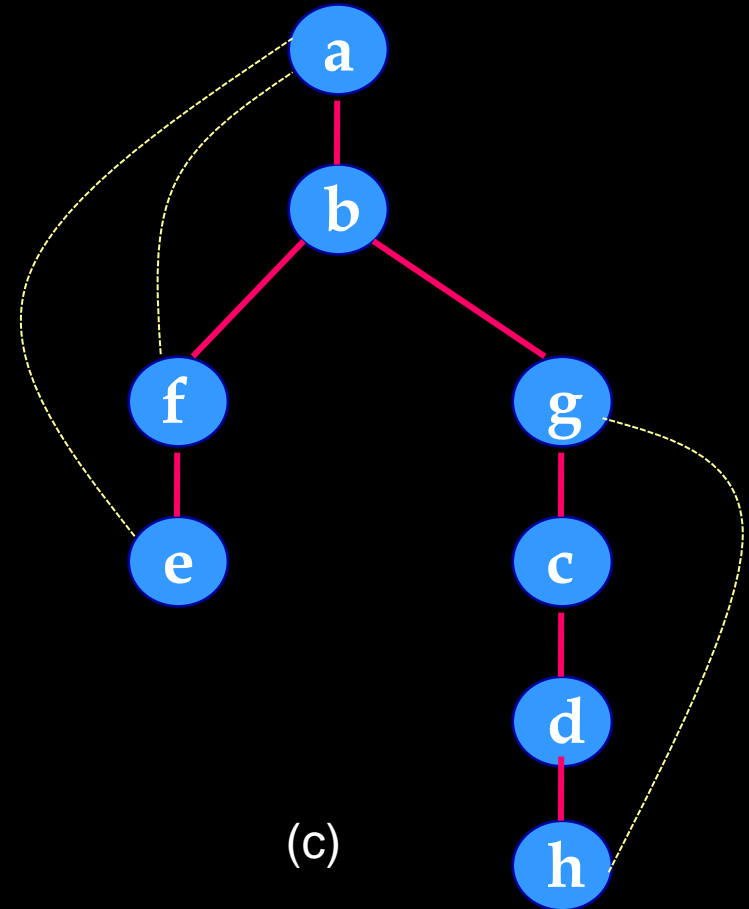
Example:



(a)

$a_{1,8}$
 $b_{2,7}$
 $f_{3,2}$
 $e_{4,1}$
 $g_{5,6}$
 $c_{6,5}$
 $d_{7,4}$
 $h_{8,3}$

(b)



(c)

 tree edge
 back edge

Figure: (a) Graph (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest

Pseudocode of DFS

ALGORITHM *DFS*(*G*)

//Implements a depth-first search traversal of a given graph

//Input: Graph $G = \langle V, E \rangle$

//Output: Graph *G* with its vertices marked with consecutive integers

//in the order they've been first encountered by the DFS traversal

mark each vertex in *V* with 0 as a mark of being "unvisited"

count \leftarrow 0

for each vertex *v* in *V* **do**

if *v* is marked with 0

dfs(*v*)

dfs(*v*)

//visits recursively all the unvisited vertices connected to vertex *v* by a path

//and numbers them in the order they are encountered

//via global variable *count*

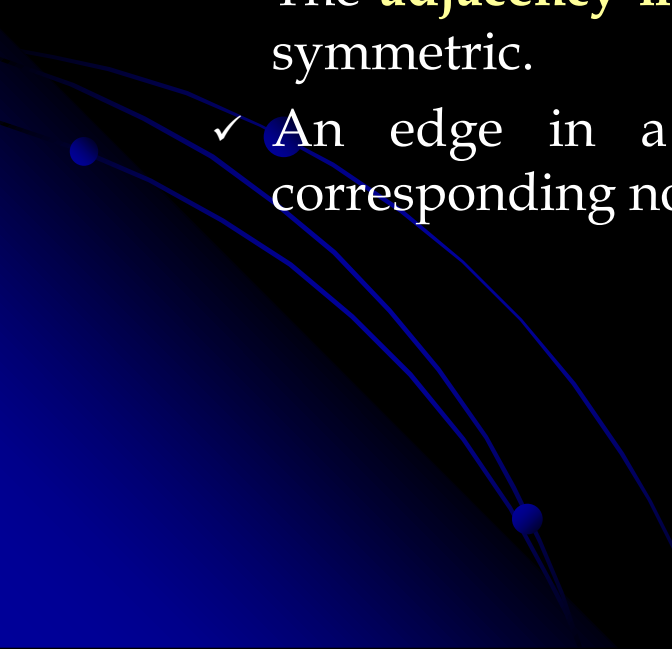
count \leftarrow *count* + 1; mark *v* with *count*

for each vertex *w* in *V* adjacent to *v* **do**

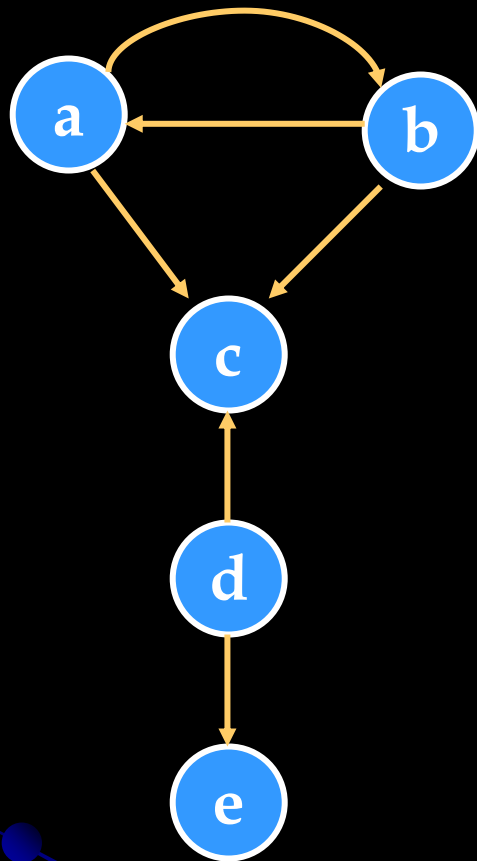
if *w* is marked with 0

dfs(*w*)

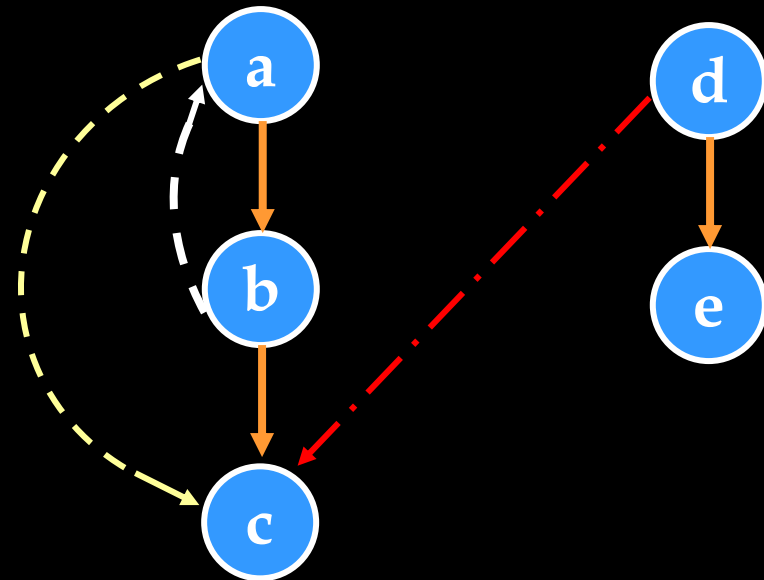
Few Basic Facts about Directed graphs

- A **directed graph**, or **digraph**, is a graph with directions specified for all its edges.
 - There are only **two notable differences** between undirected and directed graphs in representing the adjacency matrix and adjacency list:
 - ✓ The **adjacency matrix** of a directed graph does not have to be symmetric.
 - ✓ An edge in a directed graph has just one (not two) corresponding nodes in the digraph's **adjacency lists**.
- 

Directed graphs



(a)



(b)



Tree edge

Back edge

Forward edge

Cross edge

Figure: (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

Directed graphs

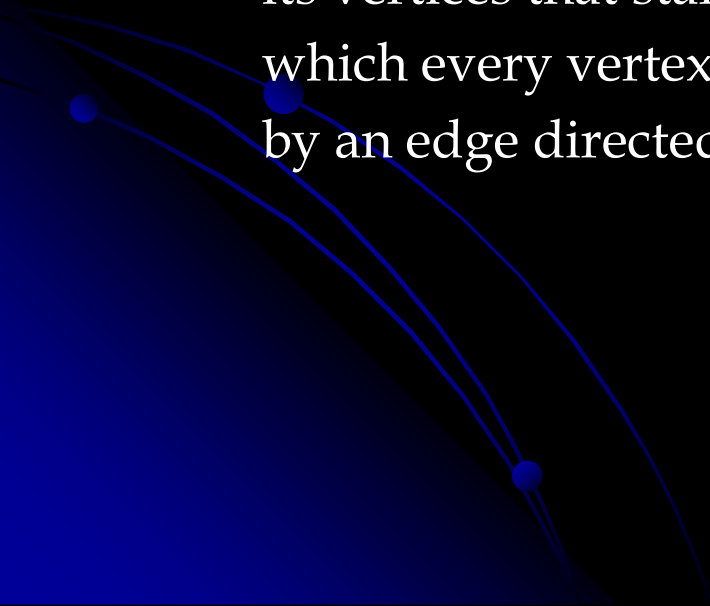
- **DFS** and **BFS** are principal traversal algorithms for traversing digraphs, but the structure of corresponding forests can be more complex.
- The DFS forest in the previous figure exhibits **all four types of edges** possible in a DFS forest of a directed graph: *tree edges* (ab, bc, de), *back edges* (ba) from vertices to their ancestors, *forward edges* (ac) from vertices to their descendants in the tree other than their children, and *cross edges* (dc), which are none of the above mentioned types.

Note: A **back edge** in a DFS forest of a directed graph can connect a vertex to its parent.

Directed graphs

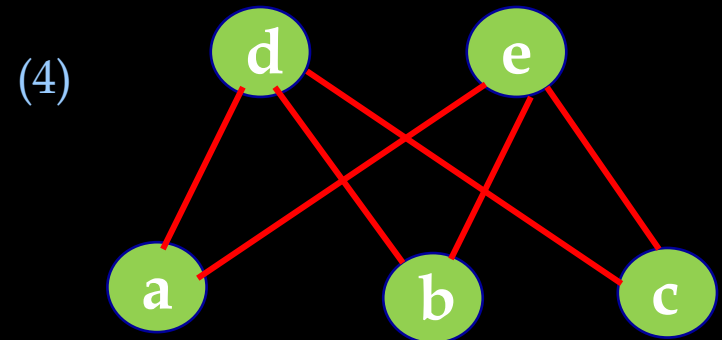
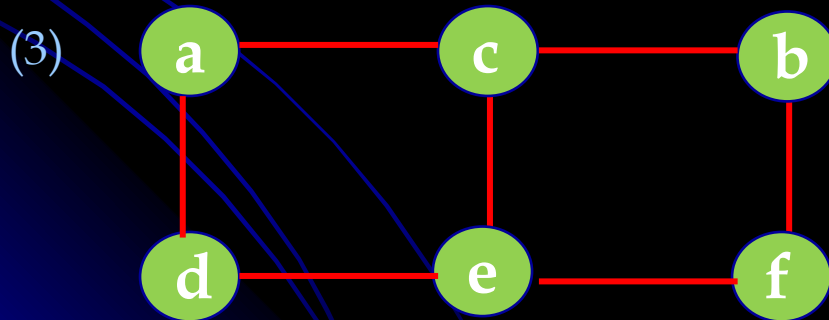
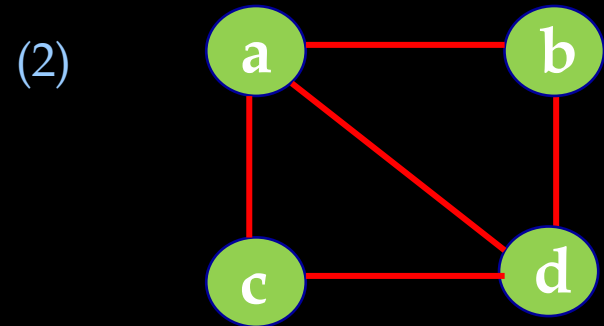
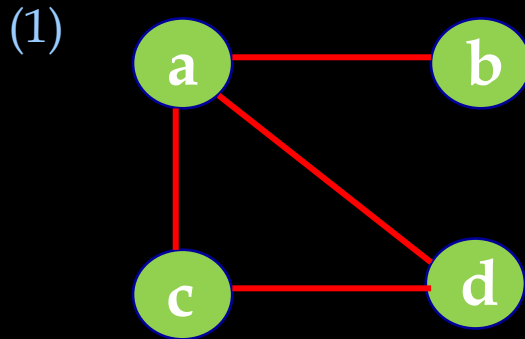
- The presence of a back edge indicates that the digraph has a **directed cycle**.
- If a DFS forest of a digraph has no back edges, the digraph is a **dag**, an acronym for **directed acyclic graph**.

Note: A **directed cycle** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.



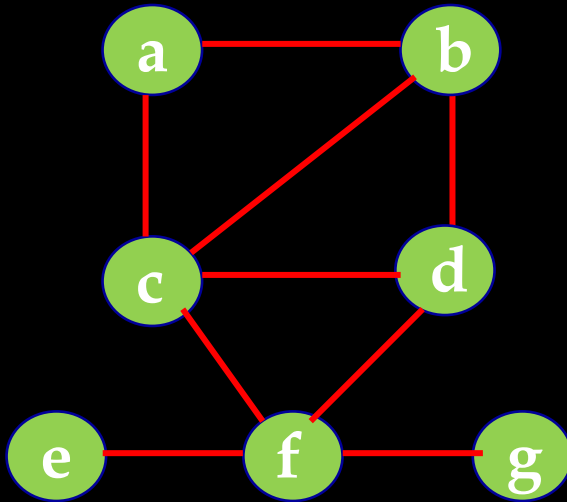
PROBLEMS

Write the DFS forest for the following graphs and also specify the order in which each vertex is pushed on to the stack and the order in which it is popped off the stack:

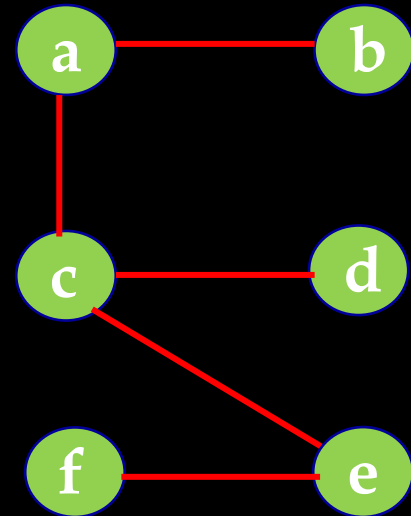


PROBLEMS

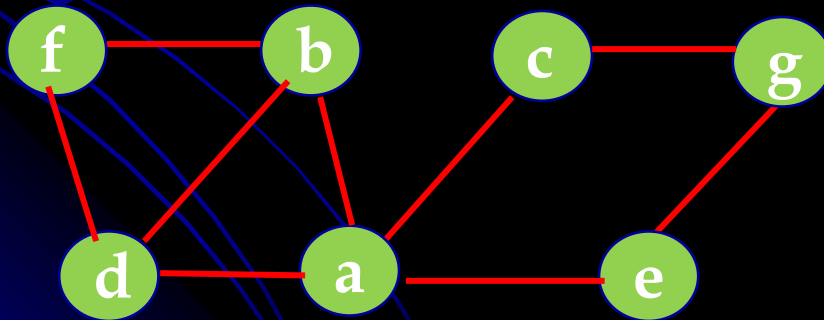
(5)



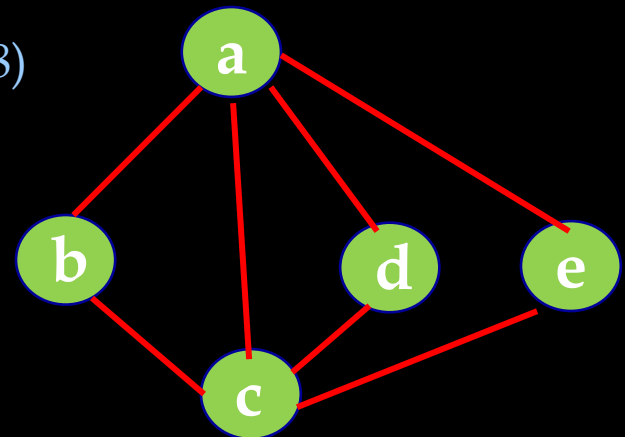
(6)



(7)

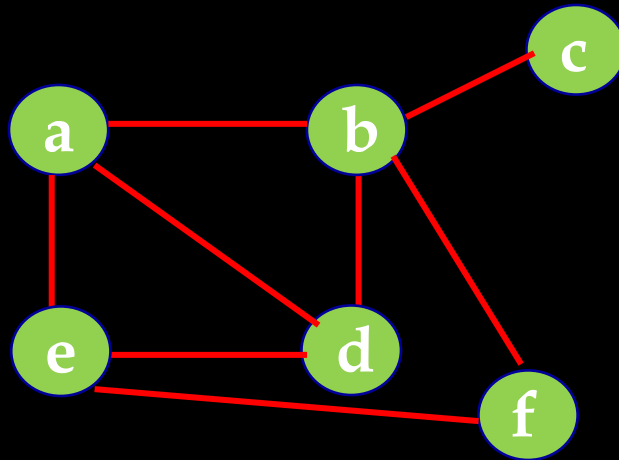


(8)

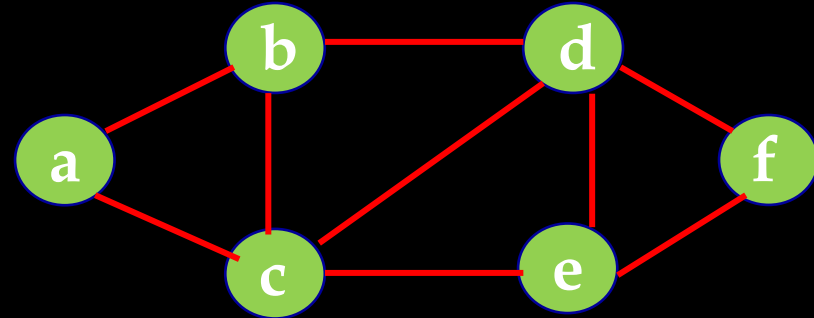


PROBLEMS

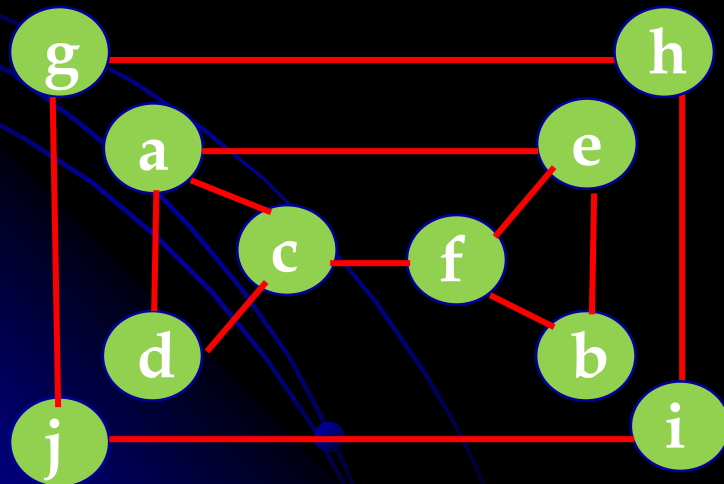
(9)



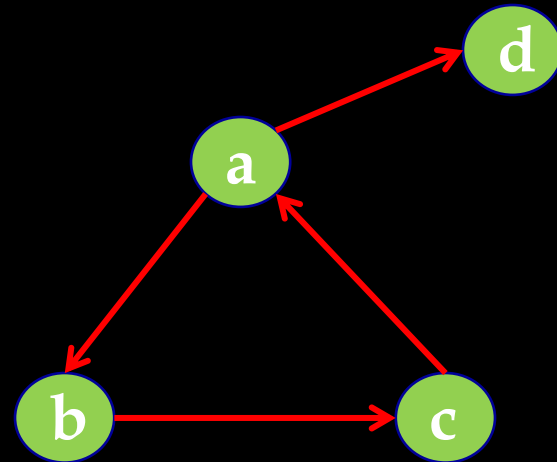
(10)



(11)

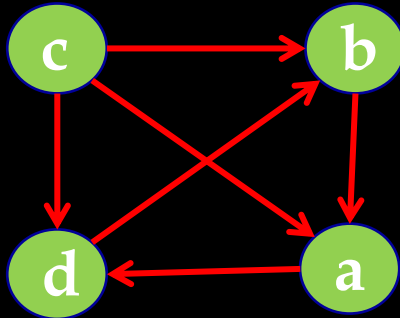


(12)

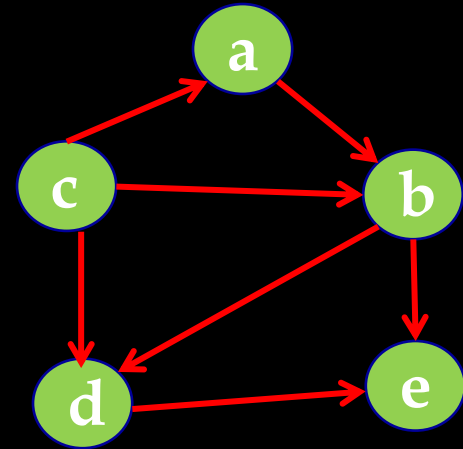


PROBLEMS

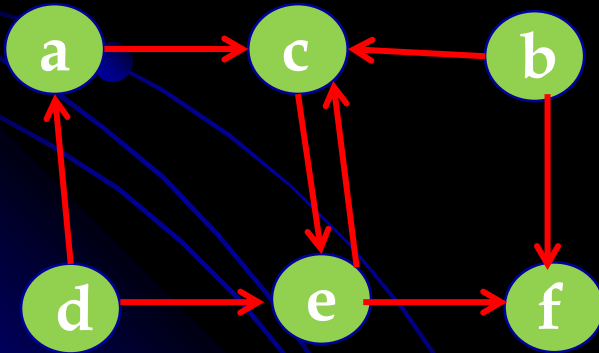
(13)



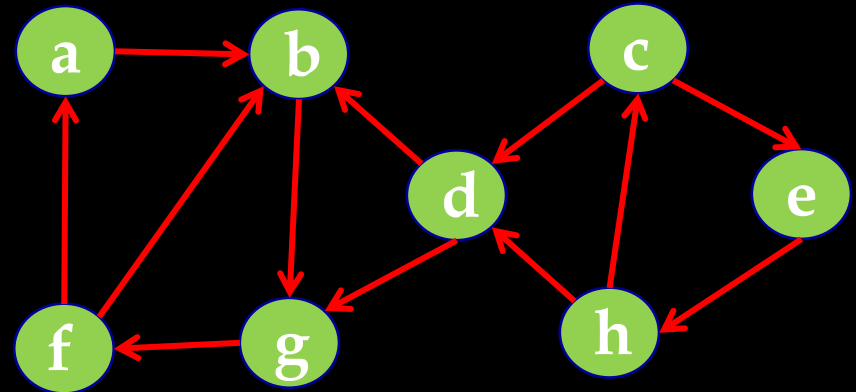
(14)



(15)

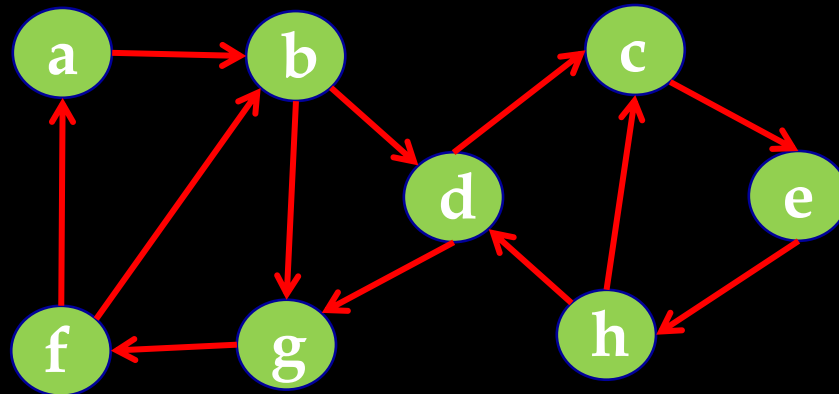


(16)

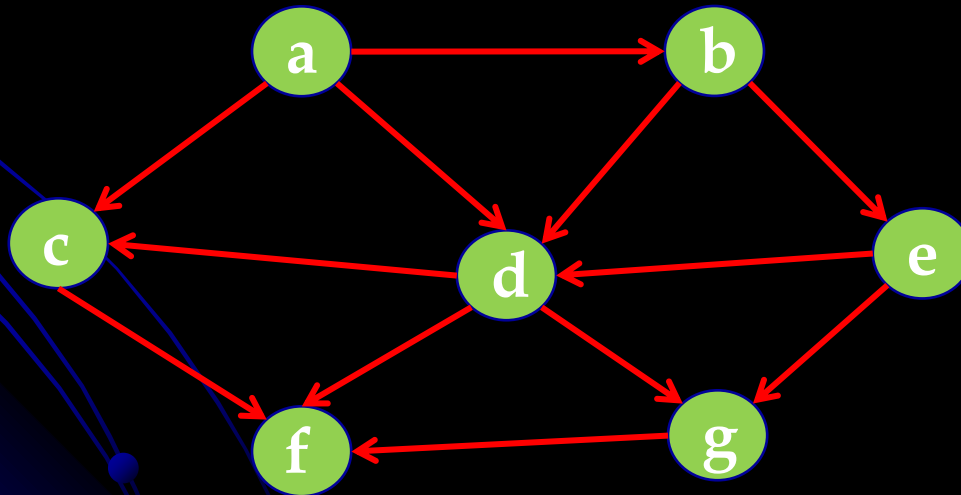


PROBLEMS

(17)

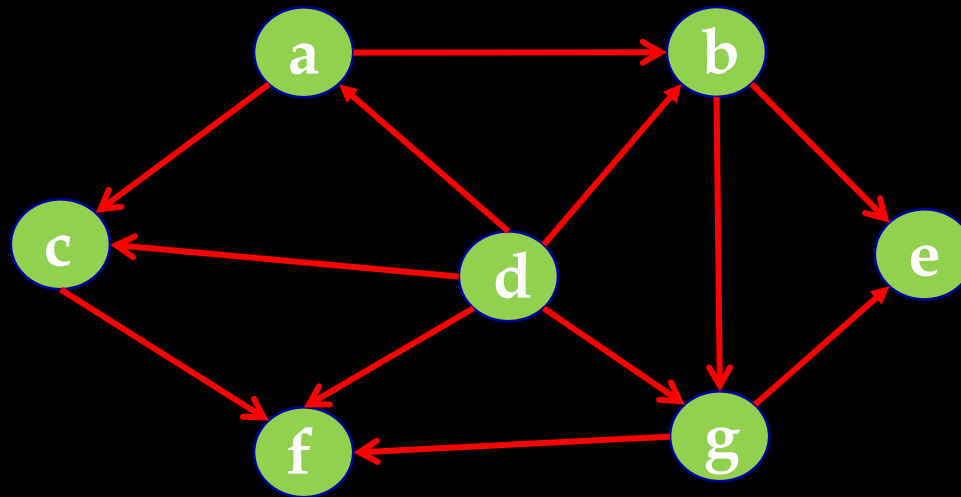


(18)

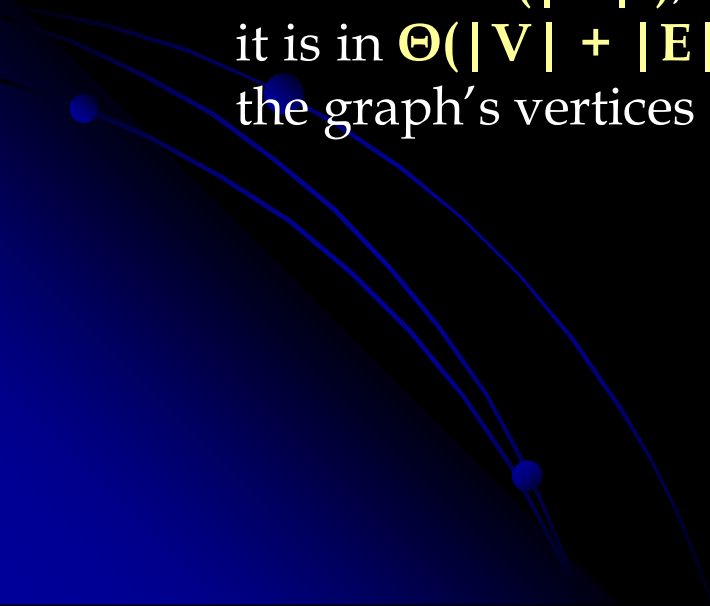


PROBLEMS

(19)



How efficient is Depth-First Search ?

- DFS algorithm is quite efficient since it takes just the time proportional to the **size of the data structure used for representing the graph** in question.
 - Thus, for the **adjacency matrix** representation, the traversal's time is in $\Theta(|V|^2)$, and for the **adjacency list** representation, it is in $\Theta(|V| + |E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.
- 

Depth-First Search

- We can look at the DFS forest as the given graph with its edges classified by the DFS traversal into **two disjoint classes**: tree edges and back edges.
 - **tree edges** are edges used by the DFS traversal to reach previously unvisited vertices.
 - **back edges** connect vertices to previously visited vertices other than their immediate predecessors in the traversal.
- DFS yields two distinct **orderings of vertices**:
 - order in which the vertices are reached for the first time (pushed onto stack).
 - order in which the vertices become dead-ends (popped off stack).

These orders are qualitatively different , and various applications can take advantage of either of them.

Applications of DFS

- **Checking connectivity:**

Since DFS halts after visiting all the vertices connected by a path to the starting vertex, checking a graph's connectivity can be done as follows: Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the graph's vertices will have been visited. If they have, the graph is connected; otherwise, it is not connected.

- **Identifying connected components of a graph.**

- **Checking acyclicity:**

If the DFS forest does not have back edges, then it is clearly acyclic.

Applications of DFS

➤ Finding articulation points of a graph:

A vertex of a connected graph is said to be its **articulation point** if its removal with all edges incident to it breaks the graph into disjoint pieces.

In DFS forest, a vertex ' u ' is an **articulation point** if one of the following two conditions is true:

- u is root of DFS forest and it has at least two children.
- u is not root of DFS forest and it has a child ' v ' such that no vertex in subtree rooted with v has a back edge to one of the ancestors of u .

Note: *Leaf* is never an articulation point.

Example to identify articulation point from DFS Forest

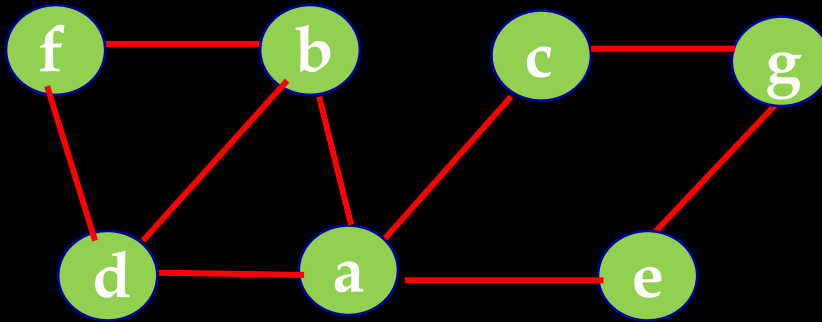
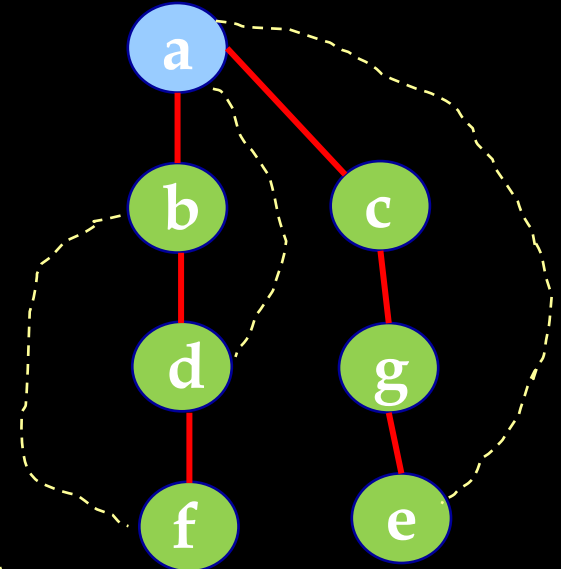
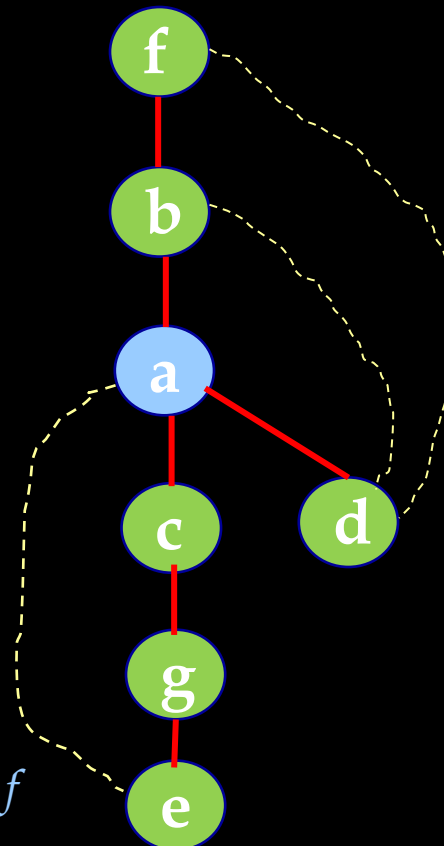


Figure: Graph G

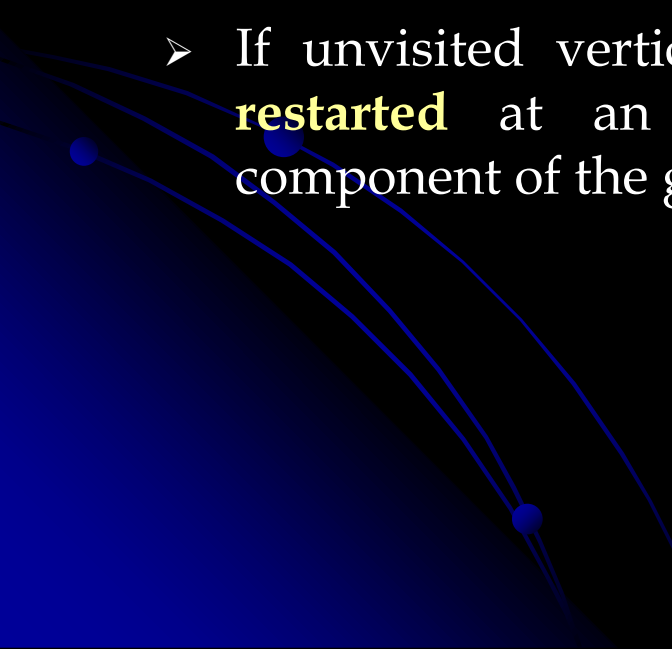


DFS traversal for graph G from vertex a



DFS traversal for graph G from vertex f

Breadth-first search (BFS)

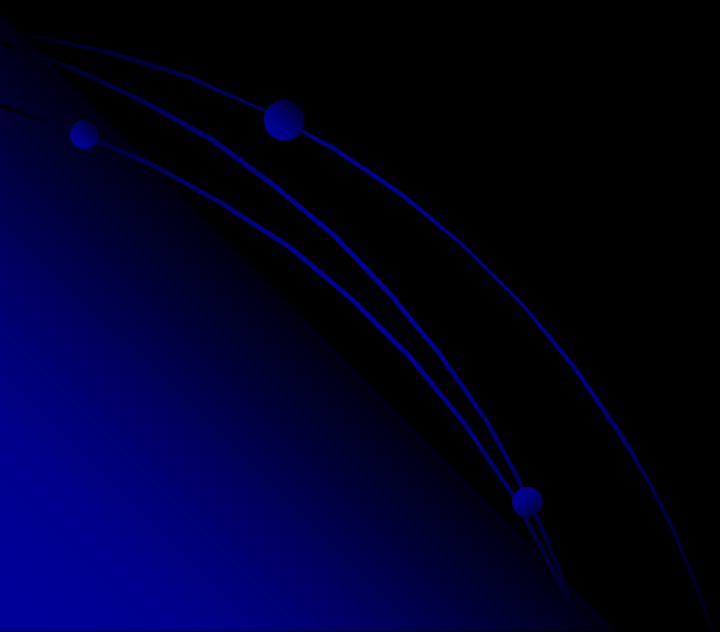
- It proceeds in a concentric manner by visiting first **all the vertices that are adjacent to a starting vertex**, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.
 - If unvisited vertices still remain, the **algorithm has to be restarted** at an arbitrary vertex of another connected component of the graph.
- 

Breadth-first search (BFS)

- Instead of a stack, **BFS uses a queue**. The queue is initialized with the **traversal's starting vertex**, which is marked as visited.
- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the **front vertex**, marks them as visited, and adds them to the queue; after that, the front vertex is **removed from the queue**.
- **Similar to level-by-level tree traversal**.
- It is useful to accompany a BFS traversal by constructing the **breadth-first search forest**. The traversal's starting vertex serves as the **root** of the first tree in such a forest.

Breadth-First Search

- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from it is being reached from with an edge called a **tree edge**.
- 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 **cross edge**.



Pseudocode of BFS

ALGORITHM *BFS*(*G*)

//Implements a breadth-first search traversal of a given graph

//Input: Graph $G = \{V, E\}$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they have been visited by the BFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

count $\leftarrow 0$

for each vertex v in V **do**

if v is marked with 0

bfs(v)

bfs(v)

//visits all the unvisited vertices connected to vertex v by a path

//and assigns them the numbers in the order they are visited

//via global variable *count*

count \leftarrow *count* + 1; mark v with *count* and initialize a queue with v

while the queue is not empty **do**

for each vertex w in V adjacent to the front vertex **do**

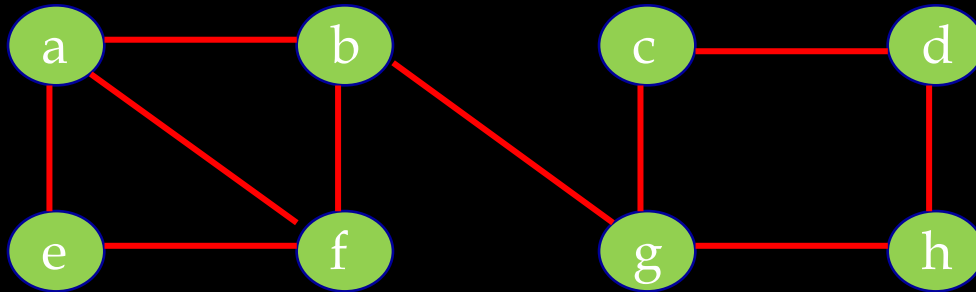
if w is marked with 0

count \leftarrow *count* + 1; mark w with *count*

 add w to the queue

 remove the front vertex from the queue

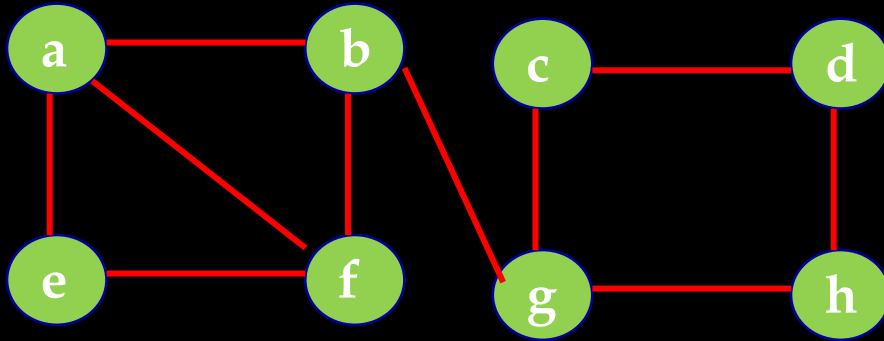
Example1 of BFS traversal of an undirected graph



1	2	3	4	5	6	7	8
a	b	e	f				
	b	e	f	g			
		e	f	g			
			f	g			
				g	c	h	
					c	h	d
						h	d
							d

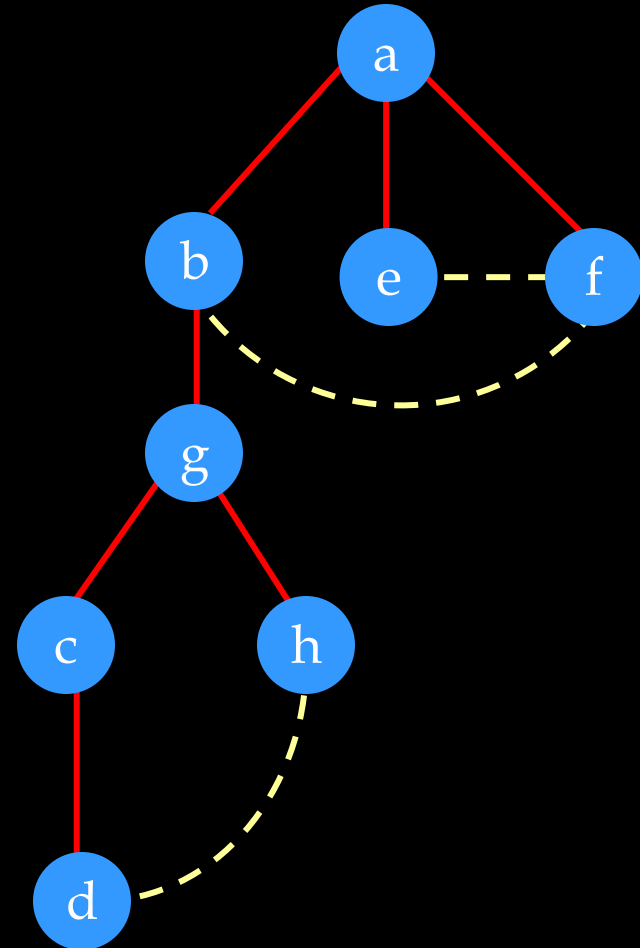
BFS traversal queue:

Example1 of BFS traversal of an undirected graph

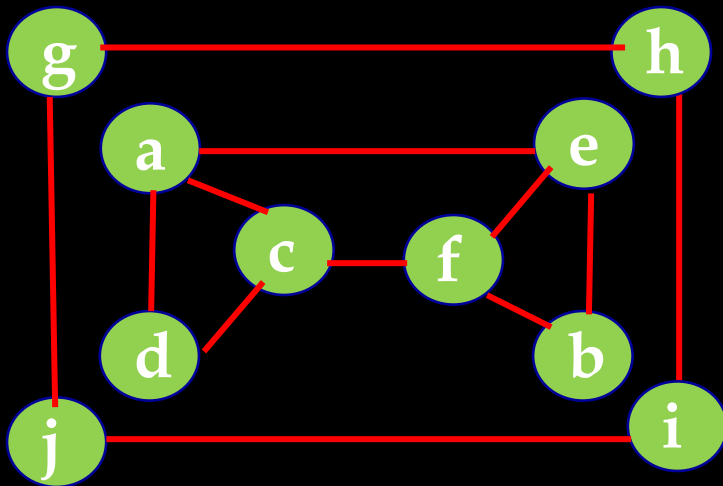


— tree edge
- - - cross edge

BFS tree:

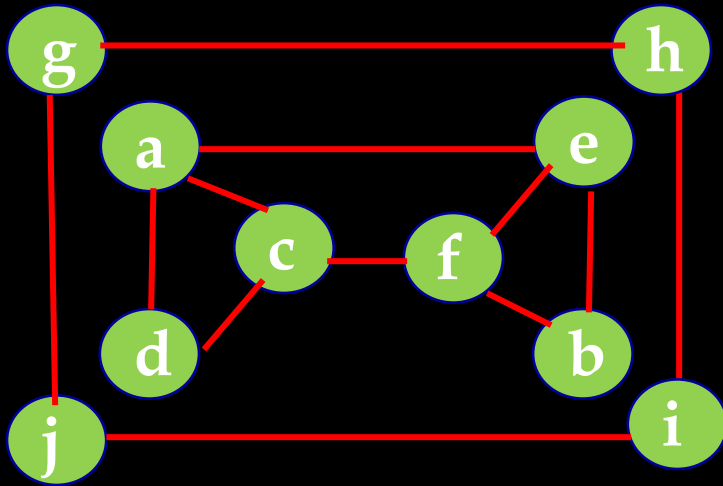


Example2 of BFS traversal of an undirected graph

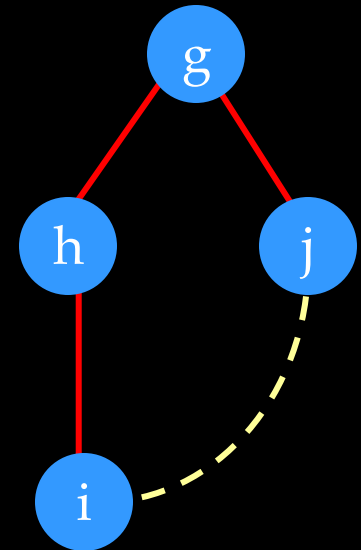
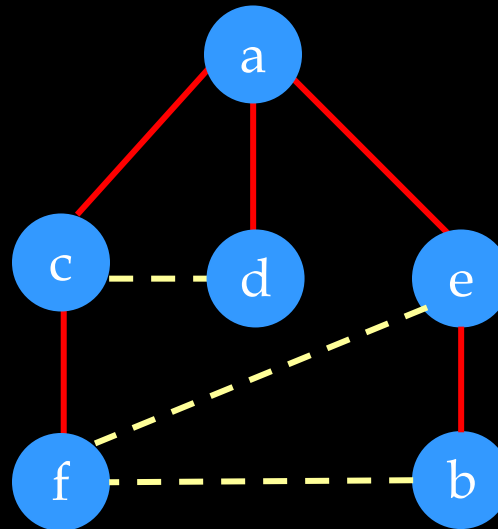
[illegible]

BFS traversal queue:

Example2 of BFS traversal of an undirected graph



BFS Traversal:



tree edge

cross edge

Breadth-First Search

- BFS has **same efficiency** as DFS :
 - $\Theta(|V|^2)$ for the adjacency matrix representation
 - $\Theta(|V| + |E|)$ for the adjacency list representation.
- BFS yields a **single ordering of vertices** because the queue is a FIFO structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it (order added/ deleted from queue is the same).
- BFS forest of an undirected graph can have two kinds of edges:
 - **Tree edges** are the ones used to reach previously unvisited vertices.
 - **Cross edges** connect vertices to those visited before, but, unlike back edges in a DFS tree, they connect vertices either on the *same or adjacent levels of a BFS tree*.

Applications of BFS

- Checking connectivity of a graph
- Checking acyclicity of a graph
- BFS can be helpful in some situations where DFS cannot.
 - For example, BFS can be used for **finding a path with the fewest number of edges between two given vertices.**
 - We start a BFS traversal at one of the two vertices given and stop it as soon as the other vertex is reached.
 - The simple path from the root of the BFS tree to the second vertex is the path sought.

Application of BFS: Finding a minimum-edge path

Path **a-b-c-g** in the below graph has the fewest number of edges among all the paths between vertices **a** and **g**.

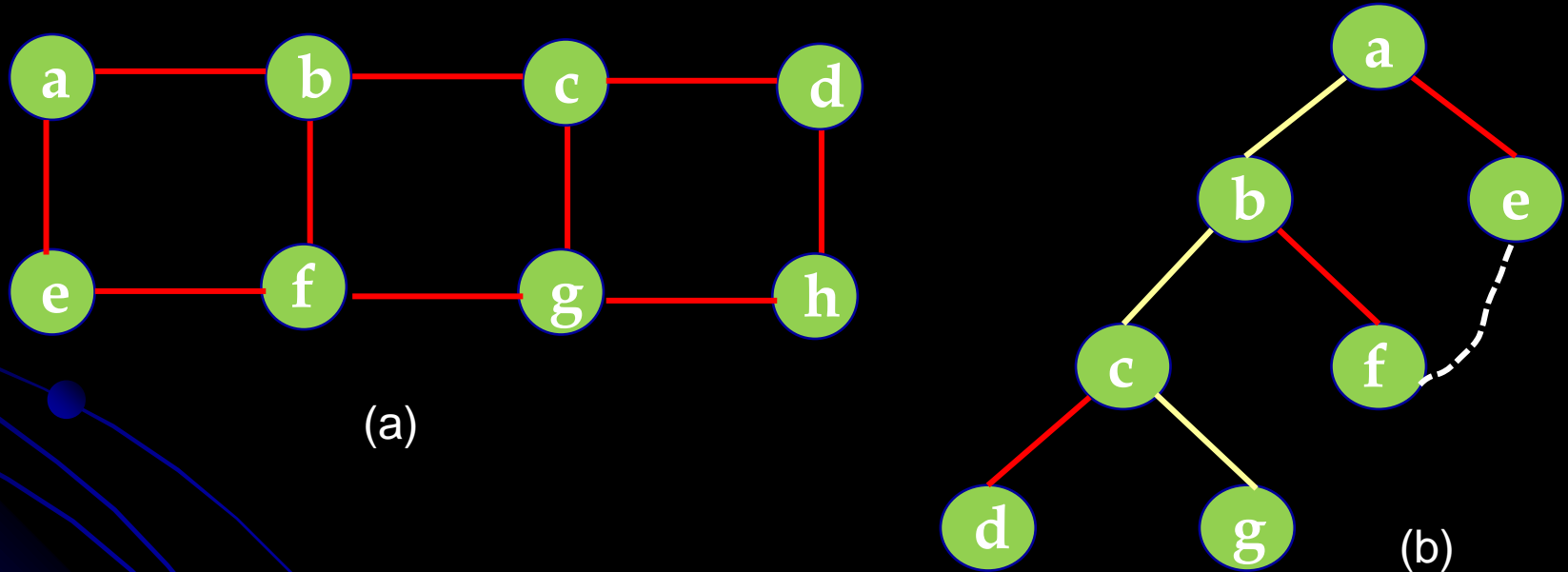


Figure: Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

Main facts about DFS and BFS

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(V ^2)$	$\Theta(V ^2)$
Efficiency for adjacency lists	$\Theta(V + E)$	$\Theta(V + E)$

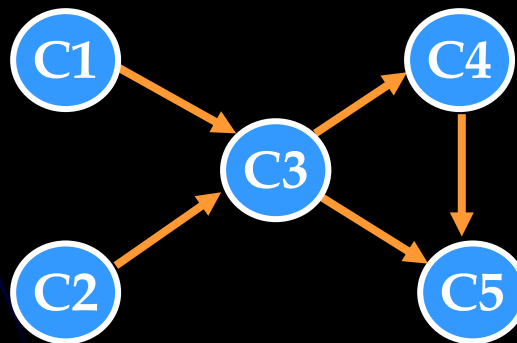
Topological Sorting

EXAMPLE:

- Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.
- The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term.

In which order should the student take the courses?

- The above situation can be modeled by a digraph in which vertices represents courses and directed edges indicate prerequisite requirements.



Topological Sorting

- The question is whether we can list the vertices of a digraph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. This problem is called topological sorting.
- For topological sorting to be possible, a digraph must be a **dag**. i.e., if a digraph has no cycles, the topological sorting problem for it **has a solution**.
- ➤ There are two efficient algorithms that both verify whether a digraph is a dag, and if it is a **dag** then it produces an **ordering of vertices** that solves the topological sorting problem:
 - **DFS – based algorithm**
 - **Source – removal algorithm**

Topological sorting Algorithms

1. DFS-based algorithm:

- DFS traversal noting the order vertices are **popped off stack** (i.e., the order in which vertices become dead ends).
- **Reverse order** solves topological sorting
- Back edges encountered? → NOT a **dag**!

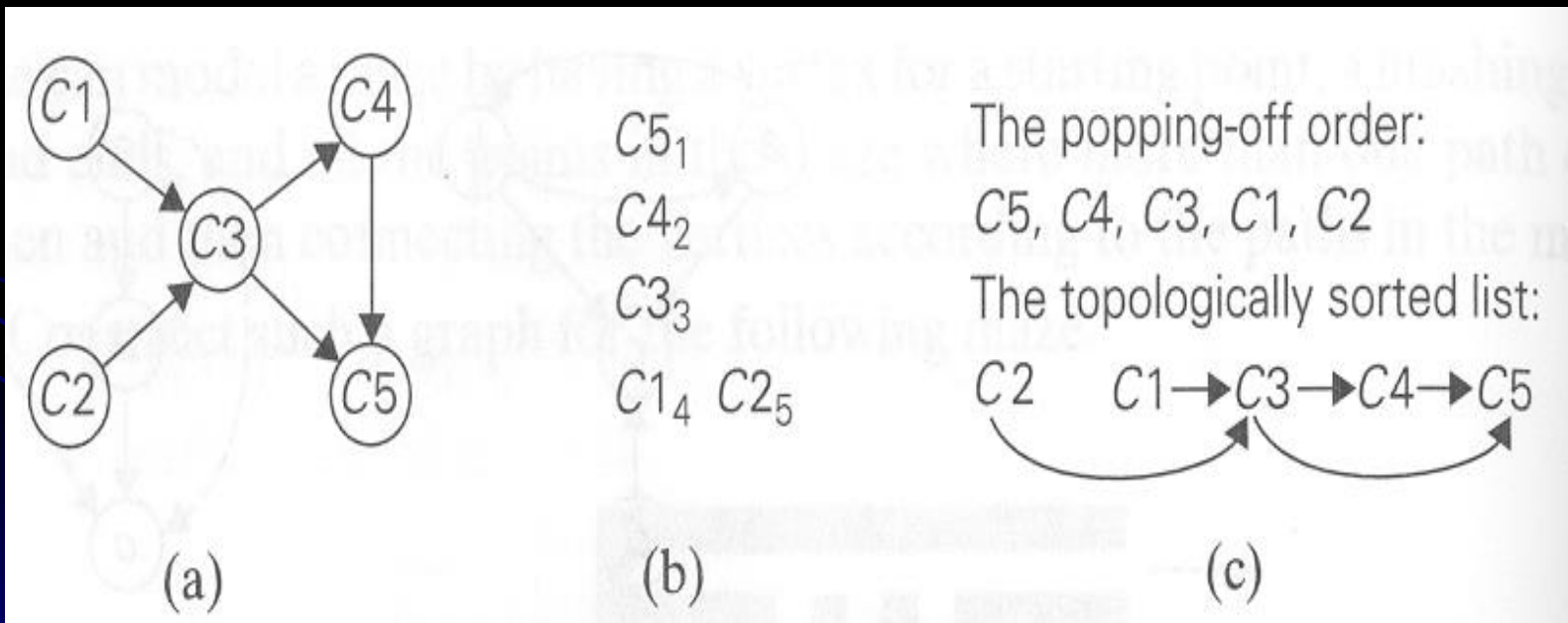
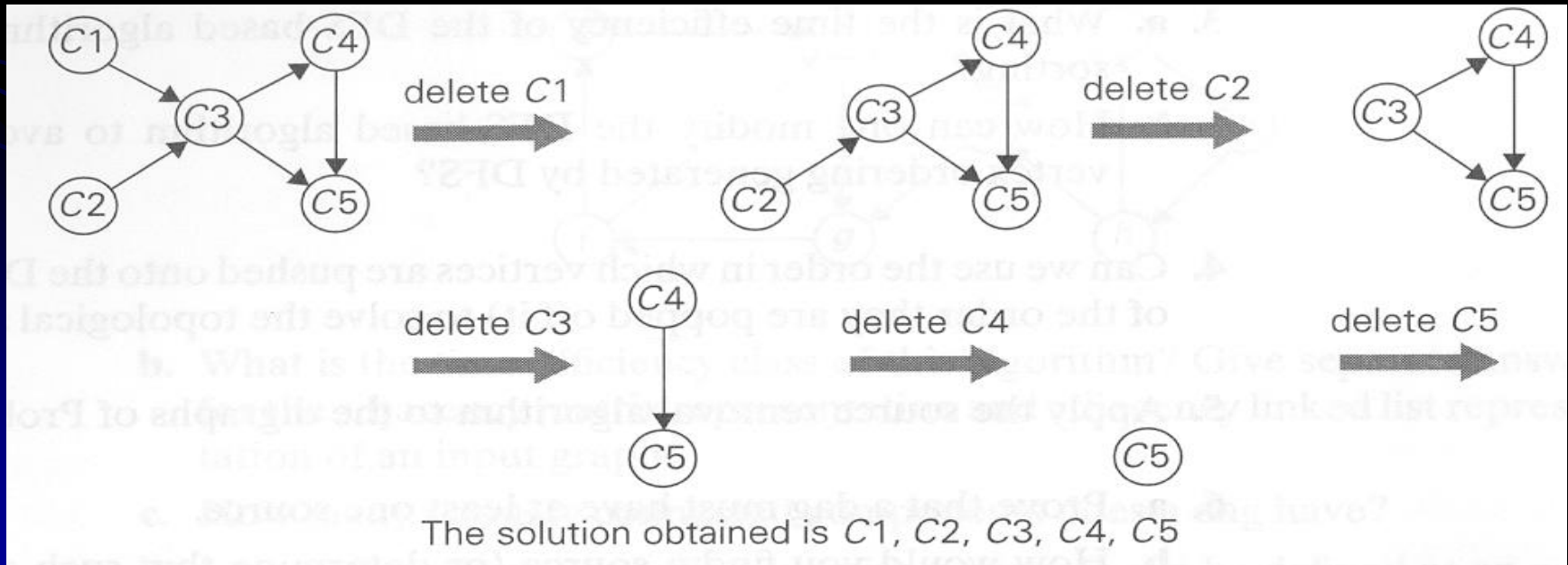


Figure: (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

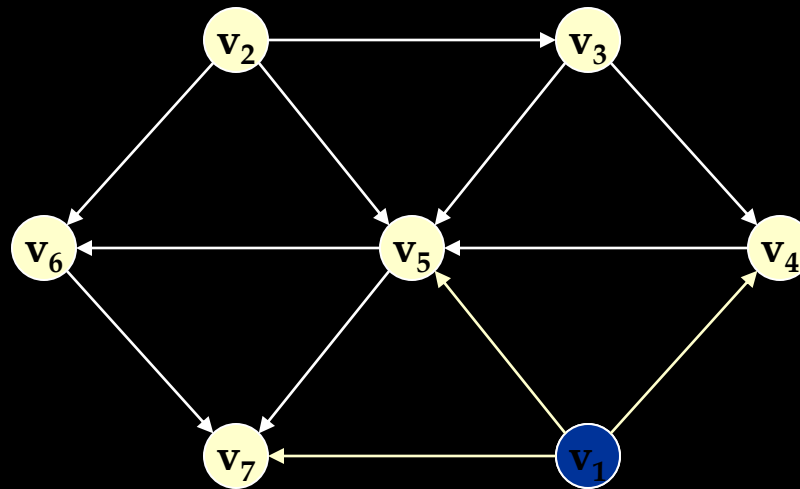
Topological sorting Algorithms

2. Source removal algorithm:

- This algorithm is based on direct implementation of the decrease(by one) – and – conquer technique: Repeatedly, identify in a remaining digraph a **source**, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.
- The **order in which the vertices are deleted** yields a solution to the topological sorting problem.

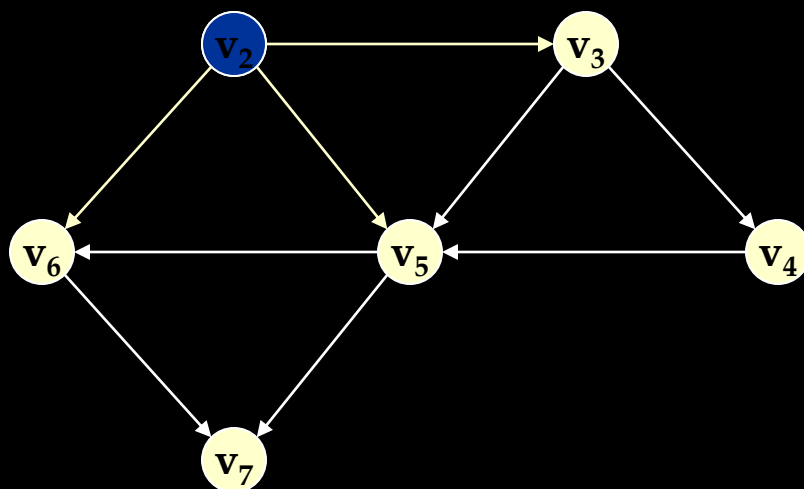


Topological Ordering: Source Removal Algorithm: Example



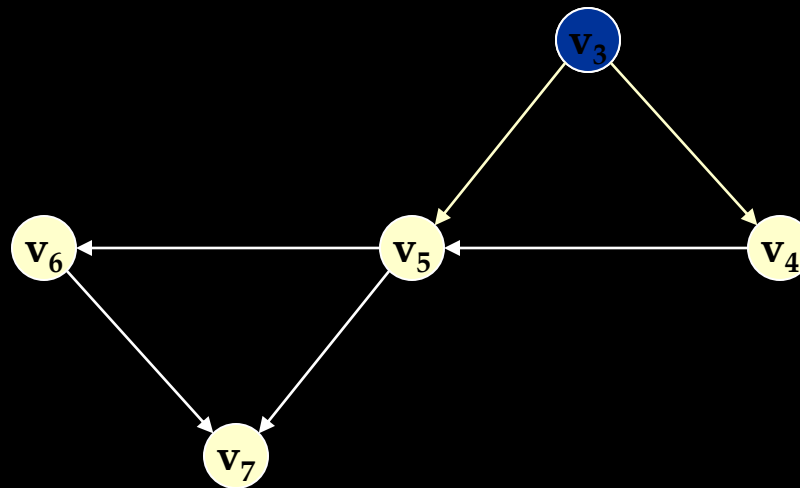
Topological order:

Topological Ordering: Source Removal Algorithm: Example



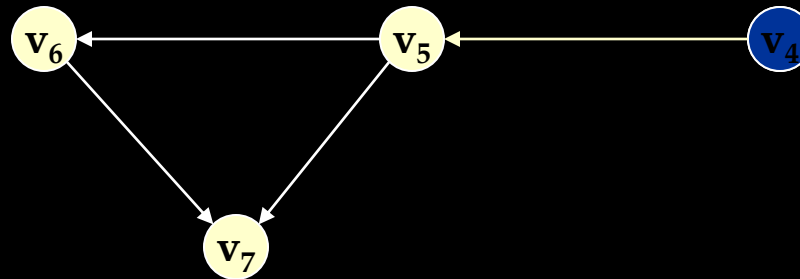
Topological order: v_1

Topological Ordering: Source Removal Algorithm: Example



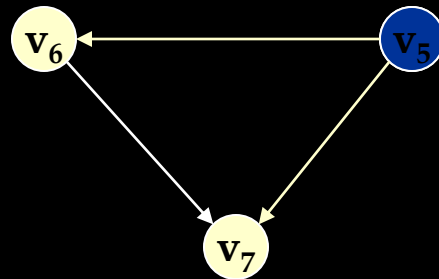
Topological order: v_1, v_2

Topological Ordering: Source Removal Algorithm: Example



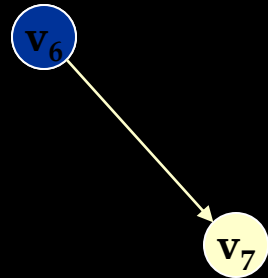
Topological order: v_1, v_2, v_3

Topological Ordering: Source Removal Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering: Source Removal Algorithm: Example



Topological order: v_1, v_2, v_3, v_4, v_5

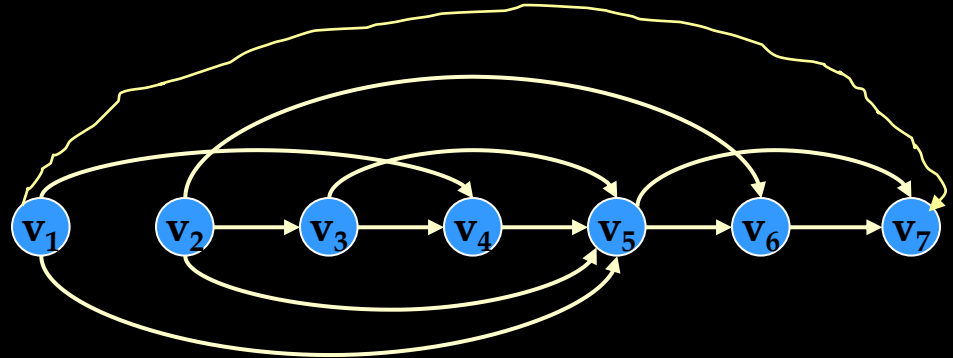
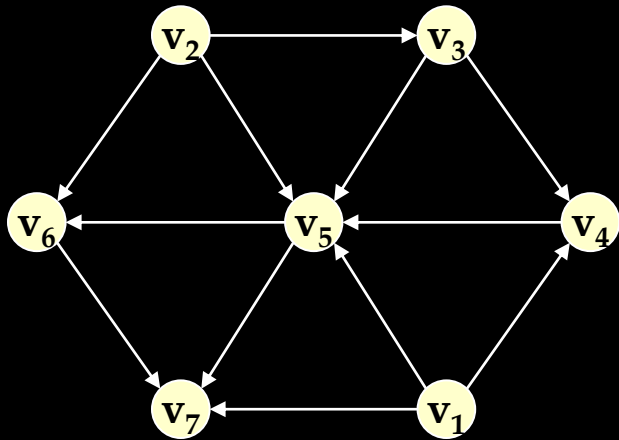
Topological Ordering: Source Removal Algorithm: Example



v_7

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering: Source Removal Algorithm: Example

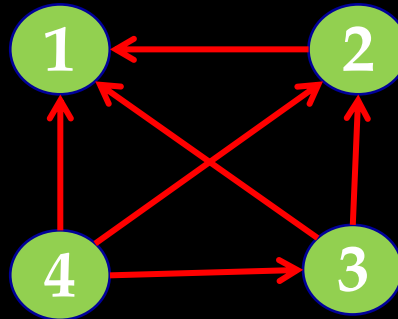


Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

PROBLEM

Solve the topological sorting problem for the following digraph using:

- a) DFS-based algorithm
- b) Source removal algorithm



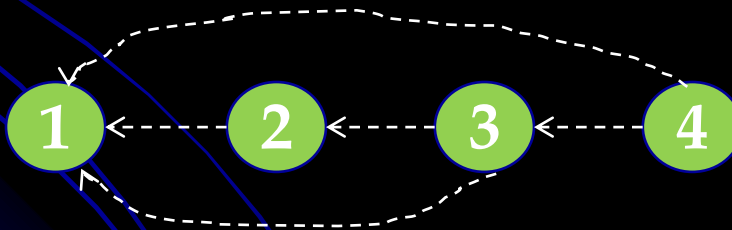
Solution:

Popping-off order:

1, 2, 3, 4

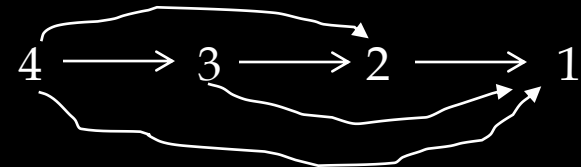
Reverse order:

4, 3, 2, 1



DFS Forest

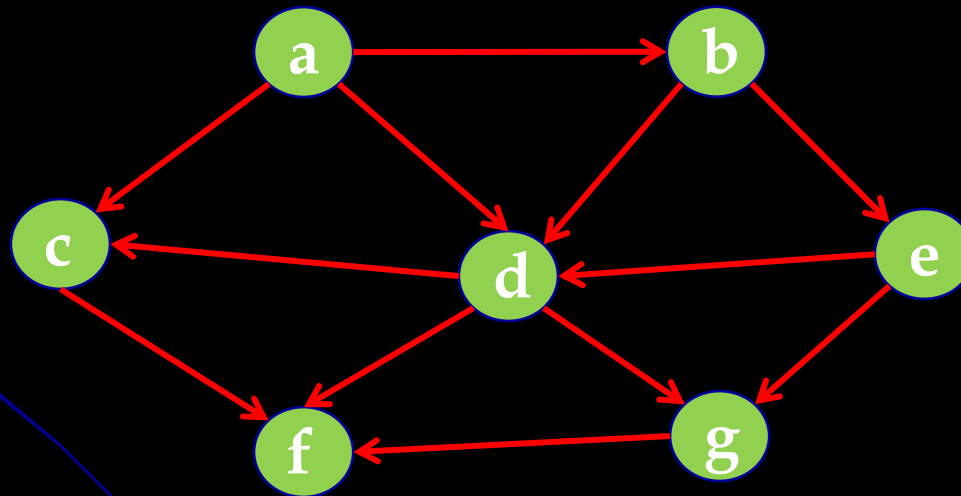
Topological order:



PROBLEM

Solve the topological sorting problem for the following digraph using:

- a) DFS-based algorithm
- b) Source removal algorithm

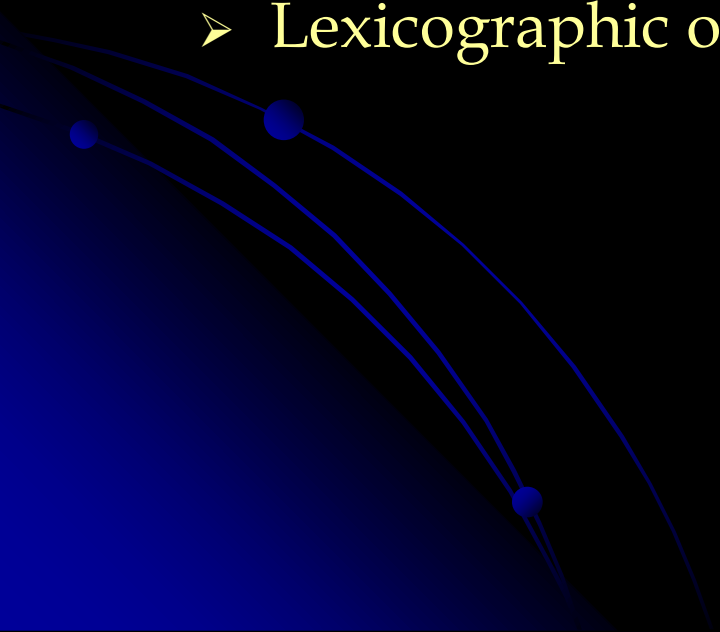


Algorithms for Generating Combinatorial Objects

- Most important types of **combinatorial objects** are:
 - permutations
 - combinations
 - subsets of a given set
- They typically arise in problems that require a consideration of **different choices**.
- **Combinatorial objects** are studied in a branch of discrete mathematics called combinatorics.
 - Mathematicians are primarily interested in different counting formulas which tell us how many items need to be generated.
- Here, we concentrate on algorithms for **generating combinatorial objects**, not just counting them.

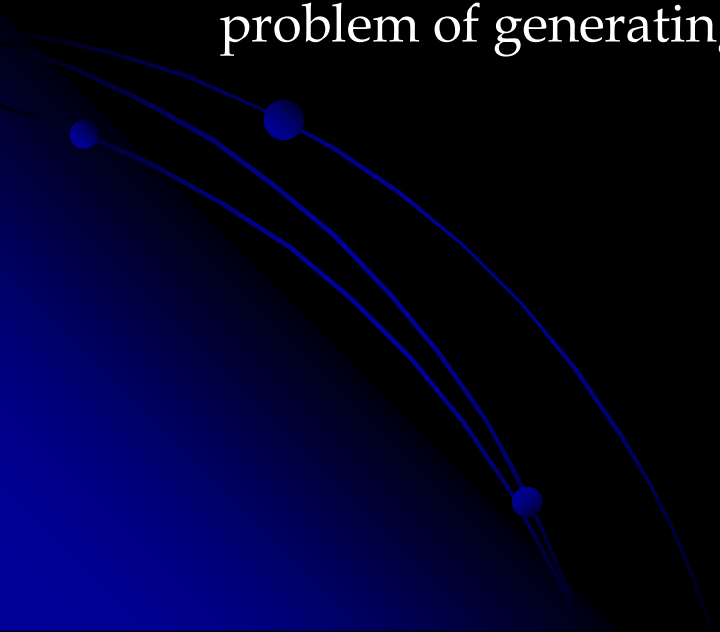
Generating Permutations

- Bottom-up minimal change algorithm
- Johnson-Trotter algorithm
- Lexicographic ordering



Generating Permutations

- For simplicity, we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n
 - They can be interpreted as indices of elements in an n -element set $\{a_1, \dots, a_n\}$
- What would the decrease-by-one technique suggest for the problem of generating all $n!$ permutations of $\{1, \dots, n\}$?



Generating Permutations

➤ Approach :

- The *smaller-by-one* problem is to generate all $(n-1)!$ permutations
- Assuming that the smaller problem is solved, we can get a solution to the larger one
 - by inserting n in each of the n possible positions among elements of every permutation of $n-1$ elements.
 - There are two possible order of insertions: either left to right or right to left.
- Total number of all permutations will be $n.(n-1)! = n!$.
Hence, we will obtain all the permutations of $\{1, \dots, n\}$.

Generating Permutations

- We can insert n in the previously generated permutations either left to right or right to left.
- We start with inserting n into $12...(n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1,...,n-1\}$ needs to be processed.
- **Minimal-change requirement** is satisfied.
 - Each permutation is obtained from the previous one by exchanging only two elements.
 - Beneficial for algorithm's speed and for applications using the permutations.

Generating Permutations

Problem:

Generate all the permutations of $n=2$ i.e., $\{1,2\}$ by the Bottom-up minimal-change algorithm.

start	1	
insert 2 into 1 right to left	12	21

Figure: Generating permutations bottom up.

Generating Permutations

Problem:

Generate all the permutations of $n=3$ i.e., $\{1,2,3\}$ by the Bottom-up minimal-change algorithm.

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

Figure: Generating permutations bottom up.

Generating Permutations

Problem:

Generate all the permutations of {1,2,3,4} by the Bottom-up minimal-change algorithm.

start	1				
insert 2 into 1 right to left	12	21			
insert 3 into 12 right to left	123	132	312		
insert 3 into 21 left to right	321	231	213		
insert 4 into 123 right to left	1234	1243	1423	4123	
insert 4 into 132 left to right	4132	1432	1342	1324	
insert 4 into 312 right to left	3124	3142	3412	4312	
insert 4 into 321 left to right	4321	3421	3241	3214	
insert 4 into 231 right to left	2314	2341	2431	4231	
insert 4 into 213 left to right	4213	2413	2143	2134	

Generating Permutations

- It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n :

- Associate a direction with each element k in a permutation
- Indicate such a direction by a small arrow written above the element

$\xrightarrow{\quad} \quad \xleftarrow{\quad} \quad \xrightarrow{\quad} \quad \xleftarrow{\quad}$
3 2 4 1

- The element k is said to be *mobile* in such an arrow-marked permutation
 - if its arrow points to a smaller number adjacent to it
 - 3 and 4 are mobile elements
 - 2 and 1 are not
- *Johnson Trotter algorithm* uses this notion of mobile element.

Generating Permutations

ALGORITHM *Johnson Trotter* (n)

// Implements Johnson-Trotter algorithm for generating
// permutations

// Input : A positive integer n

// Output : A list of all permutations of $\{1, \dots, n\}$

Initialize the first permutation with 1 2 ... n

while the last permutation has a mobile element **do**

 find its largest mobile element k

 swap k and the adjacent integer k' 's arrow points to

 reverse the direction of all the elements that are larger than k

 add the new permutation to the list

Generating Permutations

- An application of *Johnson Trotter* algorithm for $n = 2$:

$\overleftarrow{1} \overleftarrow{2}$
 $\overleftarrow{2} \overleftarrow{1}$

- This algorithm is one of the most efficient for generating permutations.
- It can be implemented to run in time proportional to the number of permutations, i.e., in $\Theta(n!)$.

Generating Permutations

- An application of *Johnson Trotter* algorithm for $n = 3$:

← ← ←
1 2 3
← ← ←
1 3 2
← ← ←
3 1 2
→ ← ←
3 2 1
← → ←
2 3 1
← ← →
2 1 3

Generating permutations

- The natural place for permutation $n \ n-1 \dots 1$ seems to be the last one on the list. This would be the case if permutations were listed in increasing order – also called the *lexicographic order*.
- The Johnson-Trotter algorithm does not produce permutations in *lexicographic order*

Example:

- Johnson-Trotter algorithm: 123, 132, 312, 321, 231, 213
- Lexicographic order: 123, 132, 213, 231, 312, 321

Lexicographic order algorithm

1. Initial permutation: a_1, a_2, \dots, a_n in increasing order.
2. Scan a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$.
3. Find the smallest element in the tail that is larger than a_i , i.e., $\min\{a_j \mid a_j > a_i, j > i\}$, and put it in position i .
4. The positions $i+1$ through n are filled with the elements a_i, a_{i+1}, \dots, a_n from which the element put in the i^{th} position has been eliminated, in increasing order.

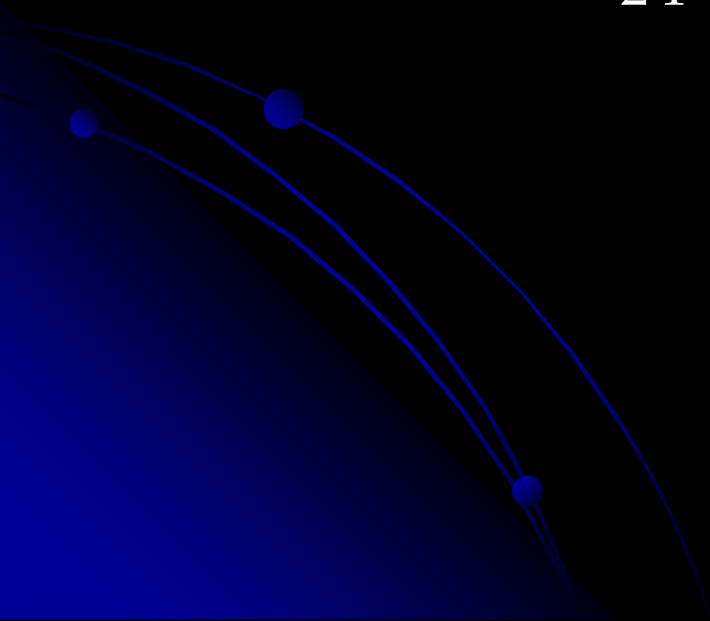
Generating permutations

Problem :

Generate all permutations of $\{1, 2\}$ by the Lexicographic-order algorithm.

Solution:

1 2
2 1



Generating permutations

Problem :

Generate all permutations of $\{1, 2, 3\}$ by the Lexicographic-order algorithm.

Solution:

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

Generating permutations

Problem 1 : Generate all permutations of $\{1, 2, 3, 4\}$ by

- a. The Johnson-Trotter algorithm.
- b. The Lexicographic-order algorithm.

Problem 2 : Generate all permutations of $\{a, b, c\}$ by

- a. Bottom-up minimal change algorithm.
- b. The Johnson-Trotter algorithm.
- c. The Lexicographic-order algorithm.

Problem 3 : Given with the current permutation as 653482, generate the next six permutations using the Lexicographic-order algorithm.

Generating subsets

Problem : Generate all 2^n subsets of a given set $A = \{a_1, \dots, a_n\}$ (i.e., power set).

➤ **Decrease-by-one approach :**

- All subsets of $A = \{a_1, \dots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do. The former group is nothing but all the subsets of $\{a_1, \dots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \dots, a_{n-1}\}$.
- Thus, once we have a list of all subsets of $\{a_1, \dots, a_{n-1}\}$, we can get all the subsets of $\{a_1, \dots, a_n\}$ by adding to the list all its elements with a_n put into each of them.

➤ An application of this algorithm to generate all subsets of $\{a_1, a_2, a_3\}$ is illustrated below:

n	subsets
0	\emptyset
1	\emptyset $\{a_1\}$
2	\emptyset $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	\emptyset $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$

Generating subsets

- We do not have to generate power sets of smaller sets.
- A convenient way of solving the problem of generating power set is based on a **one-to-one correspondence between all 2^n subsets** of an n -element set $A = \{a_1, \dots, a_n\}$ and all 2^n bit strings b_1, \dots, b_n of length n .
- The easiest way to establish such a correspondence is to assign to a subset the bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it.
- Example:
 - The bit string 000 will correspond to the empty subset of a three-element set.
 - 111 will correspond to the set itself, i.e., $\{a_1, a_2, a_3\}$
 - 110 will represent $\{a_1, a_2\}$.

Generating subsets

- With this correspondence in place, we can generate all the bit strings of length n by generating successive binary numbers from 0 to 2^n-1 .
- **Example:** For the case of $n = 3$, we obtain

bit strings	000	001	010	011	100	101	110	111
subsets	\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

- **Generation of power set in squashed order:** The order in which any subset involving a_j can be listed only after all the subsets involving a_1, \dots, a_{j-1} .

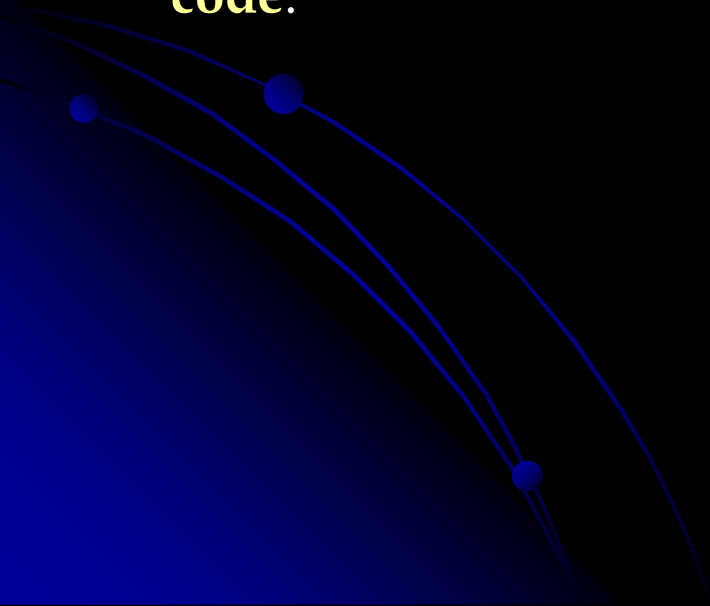
Generating subsets

- There exists a **minimal-change algorithm** for generating bit strings so that every one of them differs from its immediate predecessor by **only a single bit**.

- Example: For $n = 3$, we can get

000 001 011 010 110 111 101 100.

Such a sequence of bit strings is called the **binary reflected Gray code**.



Binary Reflected Gray code

- **n -bit Gray Codes** can be generated from list of $(n-1)$ -bit Gray codes using following steps:

1. Let the list of $(n-1)$ -bit Gray codes be L_1 . Create another list L_2 which is reverse of L_1 .
2. Modify the list L_1 by prefixing a '0' in all codes of L_1 .
3. Modify the list L_2 by prefixing a '1' in all codes of L_2 .
4. Concatenate L_1 and L_2 . The concatenated list is required list of n -bit Gray codes.

- **Example:** Following are steps for generating the 3-bit Gray code list from the list of 2-bit Gray code list.

$L_1 = \{00, 01, 11, 10\}$ (List of 2-bit Gray Codes)

$L_2 = \{10, 11, 01, 00\}$ (Reverse of L_1)

Prefix all entries of L_1 with '0', L_1 becomes $\{000, 001, 011, 010\}$

Prefix all entries of L_2 with '1', L_2 becomes $\{110, 111, 101, 100\}$

Concatenate L_1 and L_2 , we get $\{000, 001, 011, 010, 110, 111, 101, 100\}$

End of Chapter 5

