Design and Analysis of Algorithms (UE17CS251)

Unit III - Decrease-and-Conquer

Mr. Channa Bankapur
channabankapur@pes.edu

Sum of the elements of an array using
Brute Force approach.

```
Algorithm Sum(A[0..n-1])
//Sum(A[0..n-1]) = A[0] + A[1] + … + A[n-1]
//Input: Array A having n numbers
//Output: Sum of n numbers in the array A
     sum ← 0
     for i ← 1 to n
          sum ← sum + A[i]
     return sum
```

$C(n) = n$

$T(n) \in \Theta(n)$

Sum of the elements of an array using
Divide-and-Conquer approach.

```
Algorithm Sum(A[0..n-1])
//Sum(A[0..n-1]) = Sum(A[0..n/2-1]) + Sum(A[n/2..n-1])
//Input: Array A having n numbers
//Output: Sum of n numbers in the array A
     if (n = 0) return 0
     if (n = 1) return A[0]
     return    Sum(A[0..⌊(n-1)/2⌋]) +
                    Sum(A[⌊(n-1)/2⌋+1..n-1])
```

$C(n) = 2C(n/2) + 1$, $C(1) = 1$

$T(n) \in \Theta(n)$

Sum of the elements of an array using
Decrease-and-Conquer approach.

```
Algorithm Sum(A[0..n-1])
//Sum(A[0..n-1]) = Sum(A[0..n-2]) + A[n-1]
//Input: Array A having n numbers
//Output: Sum of n numbers in the array A
     if (n = 0) return 0
     return Sum(A[0..n-2]) + A[n-1]
```

$C(n) = C(n-1) + 1$, $C(1) = 1$

$T(n) \in \Theta(n)$

- **Brute Force:**
  - Sum(A[0..n-1]) = A[0] + A[1] + … + A[n-1]
  - $T(n) \in \Theta(n)$
- **Divide-and-Conquer:**
  - Sum(A[0..n-1]) = Sum(A[0..n/2-1]) + Sum(A[n/2..n-1])
  - $C(n) = 2C(n/2) + 1, C(1) = 1$

    $T(n) \in \Theta(n)$
- **Decrease-and-Conquer:**
  - Sum(A[0..n-1]) = Sum(A[0..n-2]) + A[n-1]
  - $C(n) = C(n-1) + 1, C(1) = 1$

    $T(n) \in \Theta(n)$

Finding $\mathbf{a^n}$ using **Brute Force** approach.

```
Algorithm Power(a, n)
//Finds aⁿ = a*a*...a (n times)
//Input: a ∈ R and n ∈ I+
//Output: aⁿ
     p ← 1
     for i ← 1 to n
          p ← p * a
     return p
```

$C(n) = n$

$T(n) \in \Theta(n)$

Finding $\mathbf{a^n}$ using **Divide-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds aⁿ = a^⌊n/2⌋ * a^⌈n/2⌉
//Input: a ∈ R and n ∈ I+
//Output: aⁿ
      if (n = 0) return 1
      if (n = 1) return a
      return Power(a, ⌊n/2⌋) * Power(a, ⌈n/2⌉)
```

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

$T(n) \in \Theta(n)$

Finding $\mathbf{a^n}$ using **Decrease-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds a^n = a^(n-1) * a
//Input: a ∈ R and n ∈ I+
//Output: a^n
     if (n = 0) return 1
     return Power(a, n-1) * a
```

$C(n) = C(n-1) + 1$

$T(n) \in \Theta(n)$

This approach is **Decrease-by-a-constant-and-Conquer.**
Can we solve it by **Decrease-by-a-constant-factor-and-Conquer?**

Finding $\mathbf{a^n}$ using **Decrease-by-a-constant-factor-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds aⁿ = (a⌊n/2⌋)² * aⁿ ᵐᵒᵈ ²
//Input: a ∈ R and n ∈ I+
//Output: aⁿ
      if (n = 0) return 1
      p ← Power(a, ⌊n/2⌋)
      p ← p * p
      if (n is odd) p ← p * a
   return p
```

$C(n) = C(n/2) + 2$

$T(n) \in \boldsymbol{\theta(\log n)}$

Finding $\mathbf{a^n}$ using different approaches.

- Brute-Force approach in **Θ(n)**
  - $a^n = a * a * \dots a$ (n times)
- Divide-and-Conquer approach in **Θ(n)**
  - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$, $a^0=1$, $a^1=a$
- Decrease-by-a-constant-and-Conquer in **Θ(n)**
  - $a^n = a^{n-1} * a$, $a^0=1$
- Decrease-by-a-constant-factor-and-Conquer in **Θ(log n)**
  - $a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}$, $a^0 = 1$
  - $a^n = (a^{n/2})^2$ when n is even
    $a^n = a*(a^{(n-1)/2})^2$ when n is odd and
    $a^0 = 1$

Decrease-and-Conquer:

1. Reduce problem instance into a smaller instance of the same problem.
2. Solve the smaller instance.
3. Extend the solution of the smaller instance to obtain solution to the original instance.

Also referred to as ***inductive*** or ***incremental*** approach.

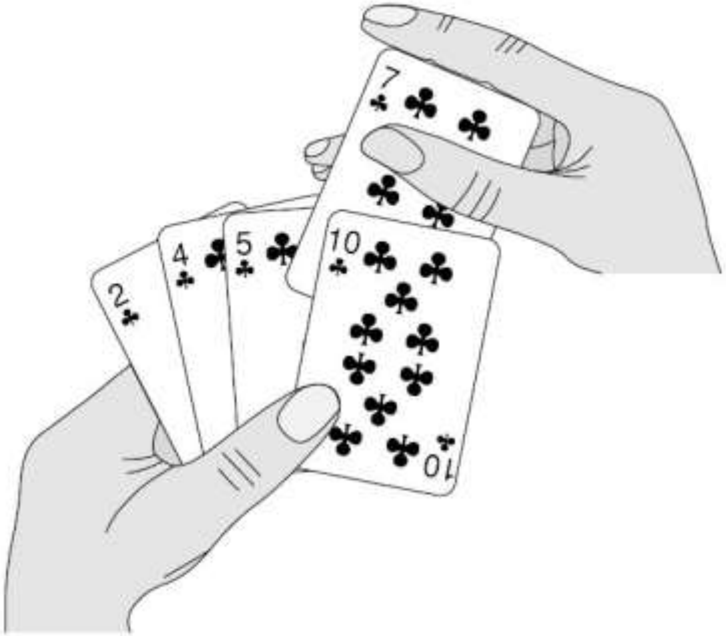
Variants of **Decrease-and-Conquer**:

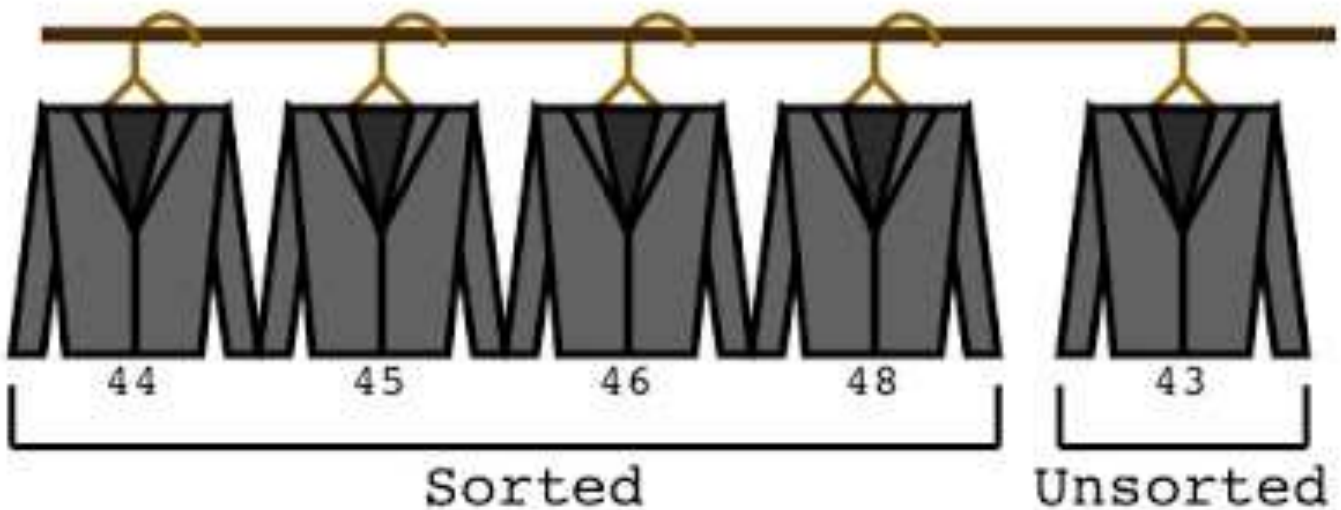
- Decrease-**by-a-constant**-and-Conquer
  - $a^n = a^{n-1} * a$
  - $Sum(a_{0..n-1}) = Sum(a_{0..n-2}) + a_{n-1}$

- Decrease-**by-a-constant-factor**-and-Conquer
  - $a^n = (a^{n/2})^2$ **when n is even**
    $a^n = a*(a^{(n-1)/2})^2$ **when n is odd**
    and
    $a^1 = a,\ a^0 = 1$
  - **Binary Search**

- Decrease-**by-variable-size**-and-Conquer
  - gcd(m, n) = gcd(n, m mod n), and gcd(m, 0) = m

Insertion Sort:





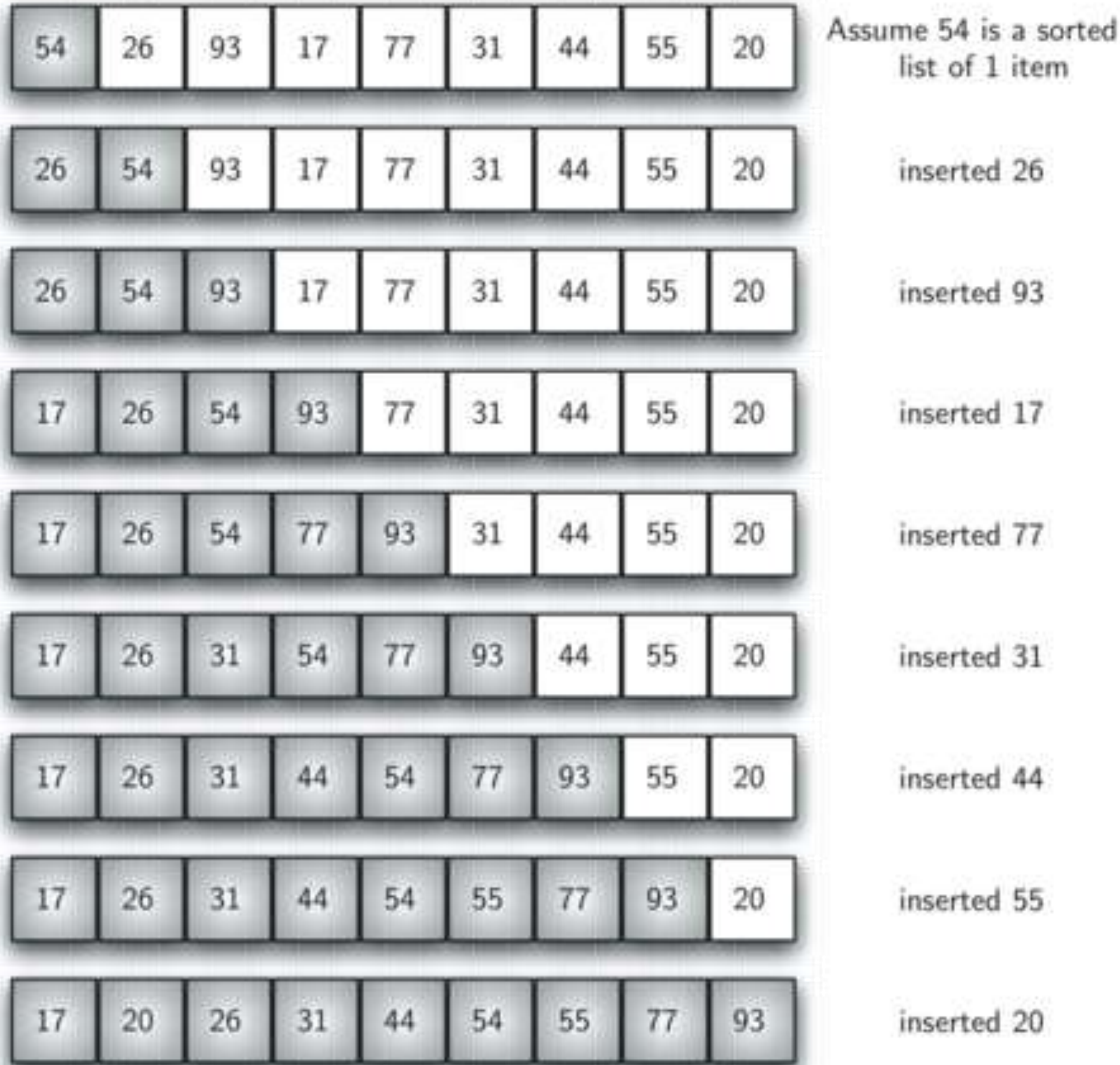
Insertion Sort: To sort an array A[0..*n*-1], sort A[0..*n*-2] and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]. It's a Decrease-by-a-constant-and-Conquer approach.

```
Algorithm InsertionSortRecursive( A[0..n-1] )
   if (n > 1)
            InsertionSortRecursive(A[0..n-2])
            temp ← A[n-1]
      j ← n-2
      while(j ≥ 0 and A[j] > temp)
            A[j+1] ← A[j]
            j ← j-1
      A[j+1] ← temp
```

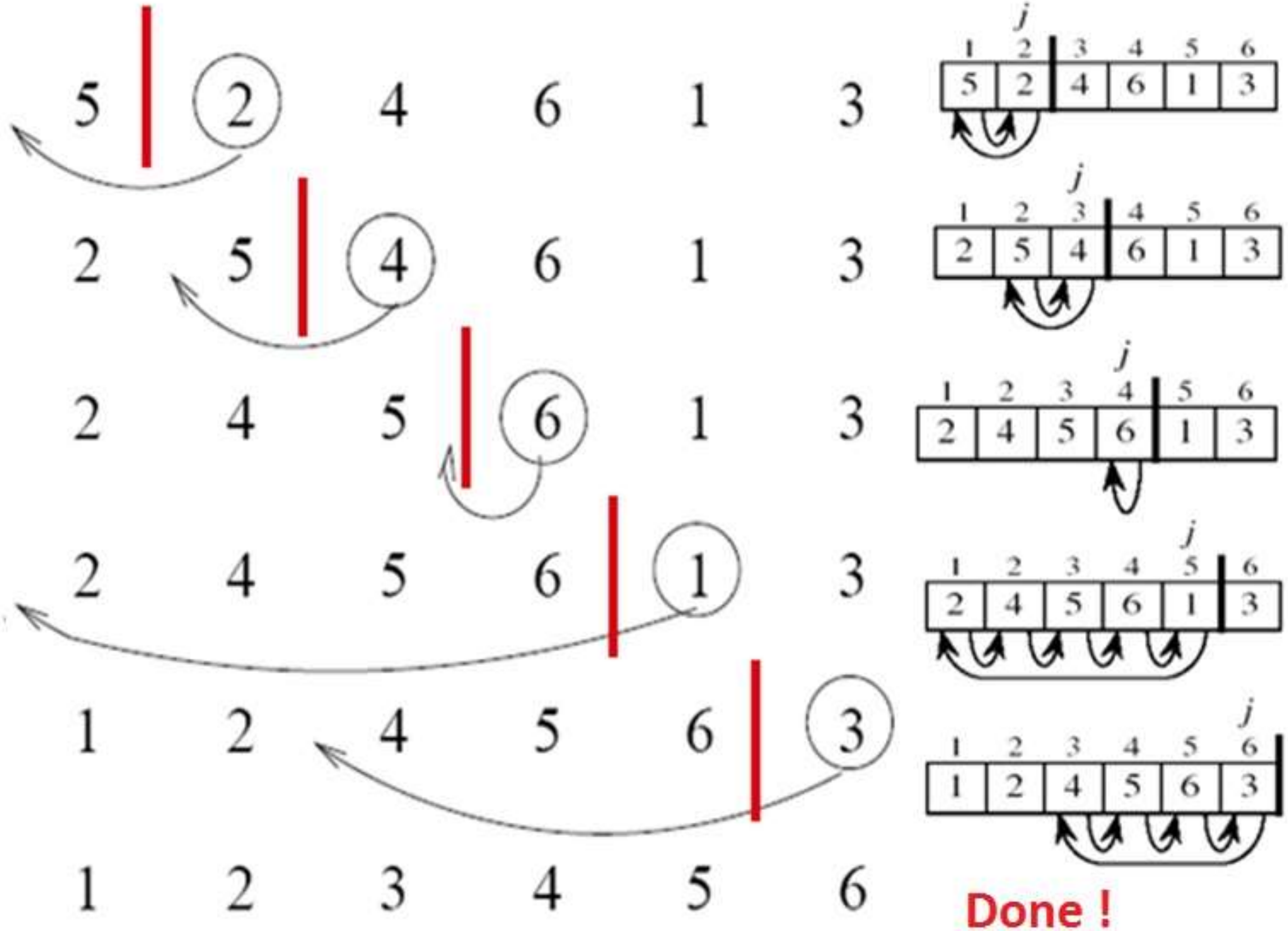
Insertion Sort: To sort an array A[0..*n*-1], sort A[0..*n*-2] and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2].

It's a Decrease-by-a-constant-and-Conquer approach.
It's usually implemented bottom-up (non-recursively).
Example: Sort 6, 4, 1, 8, 5, 5

6 **|**	<u>**4**</u>	1	8	5	5
4	6 **|**	<u>**1**</u>	8	5	5
1	4	6 **|**	<u>**8**</u>	5	5
1	4	6	8 **|**	<u>**5**</u>	5
1	4	5	6	8 **|**	<u>**5**</u>
1	4	5	5	6	8



54 26 93 17 77 31 44 55 20
Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20
inserted 26
26 54 93 17 77 31 44 55 20
inserted 93
17 26 54 93 77 31 44 55 20
inserted 17
17 26 54 77 93 31 44 55 20
inserted 77
17 26 31 54 77 93 44 55 20
inserted 31
17 26 31 44 54 77 93 55 20
inserted 44
17 26 31 44 54 55 77 93 20
inserted 55
17 20 26 31 44 54 55 77 93
inserted 20



$$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \;\big|\; A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

$$A[0] \leq \cdots \leq A[j] < A[j+1] \leq \cdots \leq A[i-1] \mid A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

```
Algorithm InsertionSort(A[0..n-1])
   for i ← 1 to n-1
          temp ← A[i]
      j ← i-1
      while(j ≥ 0 and A[j] > temp)
          A[j+1] ← A[j]
          j ← j-1
      A[j+1] ← temp
```

- **Time efficiency**
  - $C_{worst}(n) = 1 + 2 + 3 + \ldots + n - 1$

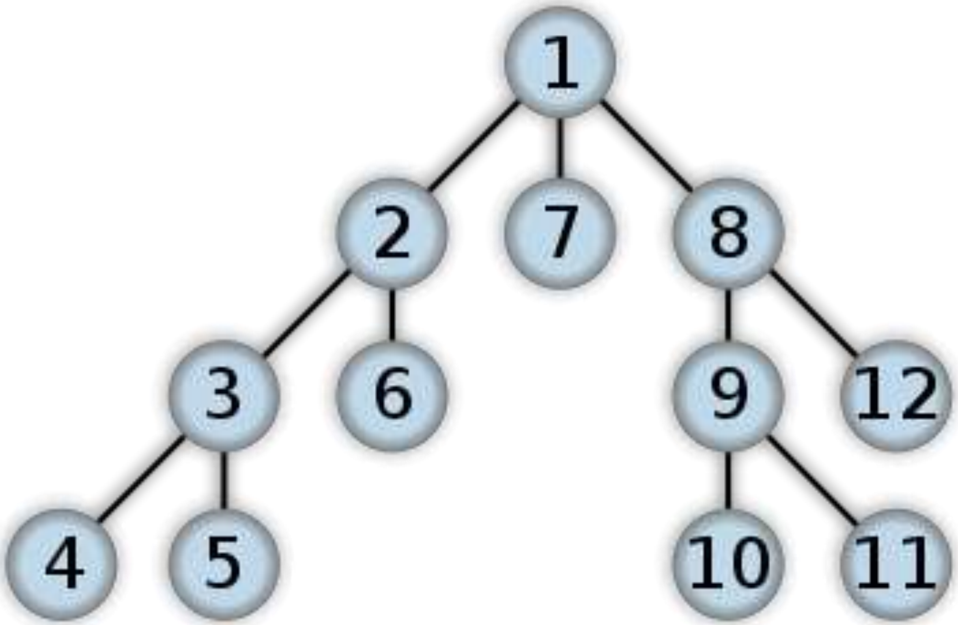
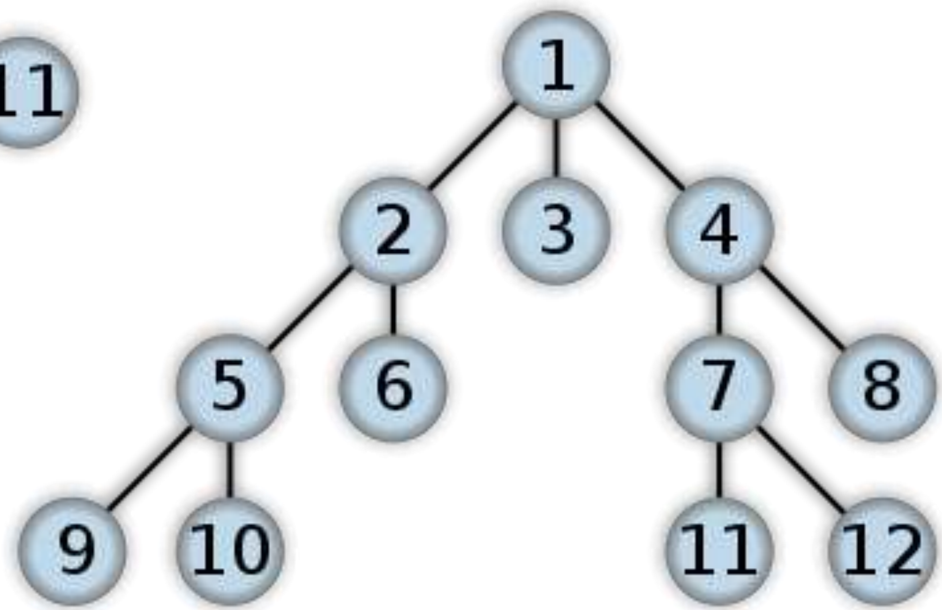
    $= n (n - 1) / 2 \in \Theta(n^2)$
  - $C_{best}(n) = n - 1 \in \Theta(n)$
  - $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$
- Fast on nearly sorted arrays
- Space complexity ?
- Stable sorting ?
- Best elementary sorting algorithm overall
  - Often used in Quicksort implementations
- Binary insertion sort?

Graph Traversal Algorithms:

Many problems require processing all graph vertices (or edges) in a systematic way.

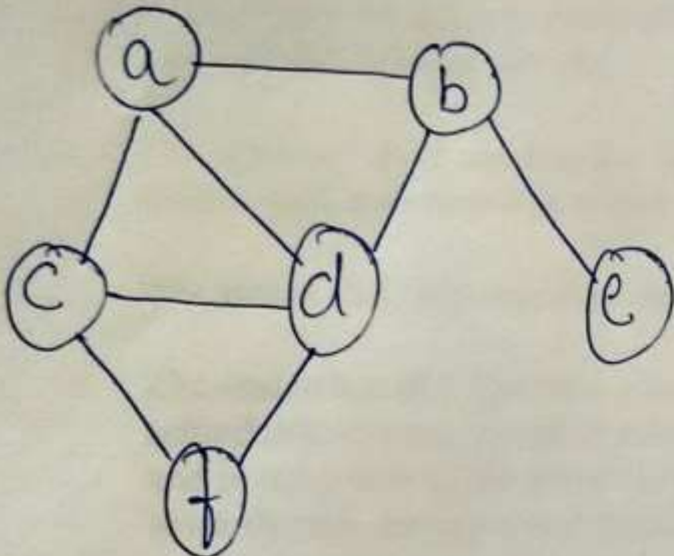
Based on the order of visiting the vertices:

- Depth-first search (DFS)
  - Search everything related a one neighbor recursively before starting with another
  - Uses Stack behavior
- Breadth-first search (BFS)
  - Search all the neighbors (first level) before the 2nd level of nodes
  - Uses Queue behavior

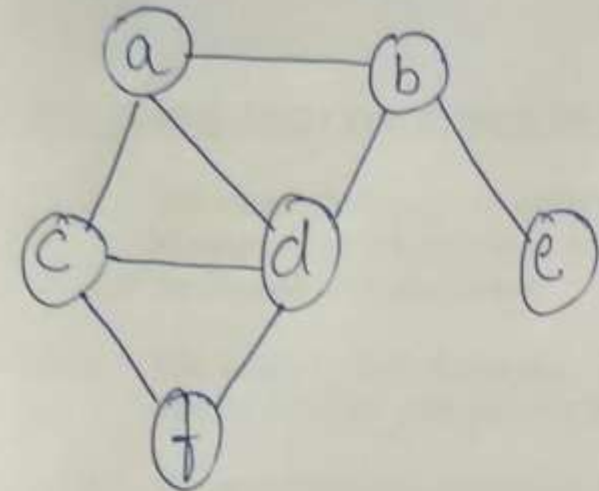

1
2
7
8
3
6
9
12
4
5
10
11

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

Algorithm dfs(v)

```
//Initially all vertices are marked with 0
    Mark v with 1
    for each vertex w in V adjacent to v
        if (w is marked with 0)
            dfs(w)
```



$$\begin{array}{cc} & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$



$$
\begin{array}{c}
\\ a \\ b \\ c \\ d \\ e \\ f
\end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\left[\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$

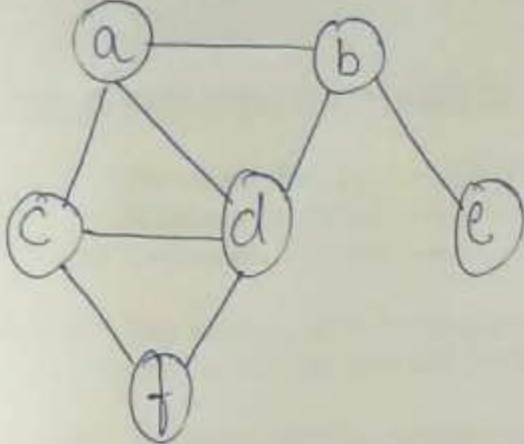
mark						
	✓					

DFS Tree

a

a_1

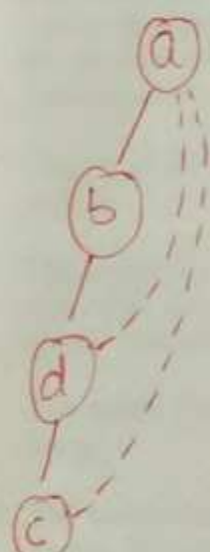
DFS stack



$$
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0
\end{bmatrix}
\end{array}
$$

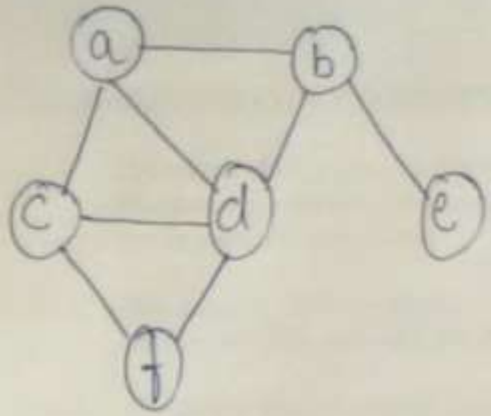
mark						
	✓	✓	✓	✓		

DFS Tree



c_4
d_3
b_2
a_1

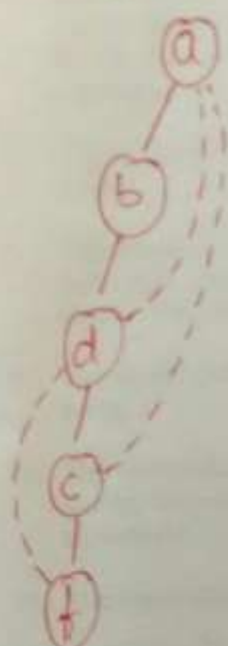
DFS Stack



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

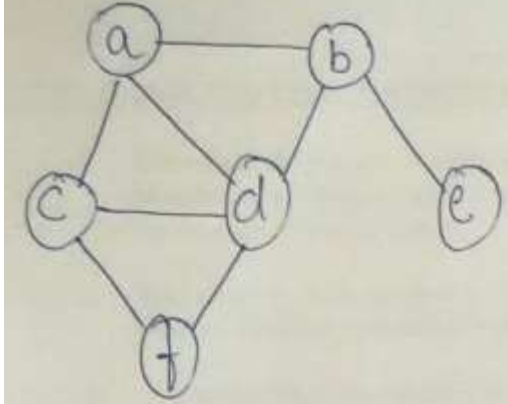
mark						
	✓	✓	✓	✓		✓

DFS Tree



$f_{5,1}$
c_4
d_3
b_2
a_1

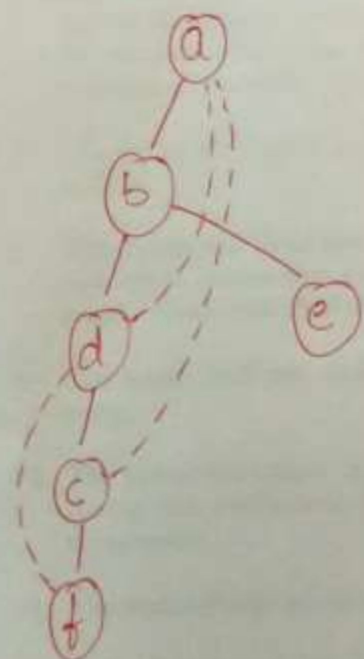
DFS Stack



$$
\begin{array}{c}
 \\ a \\ b \\ c \\ d \\ e \\ f
\end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\left[\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$

mark	✓	✓	✓	✓	✓	✓

DFS Tree



$f_{5,1}$

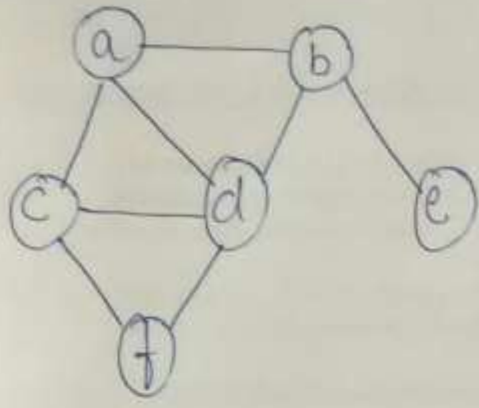
$c_{4,2}$

$d_{3,3}$ e_6

b_2

a_1

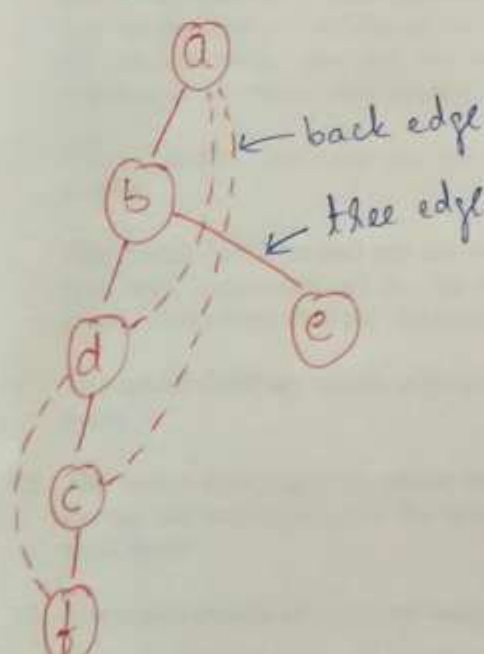
DFS Stack



$$
\begin{array}{c}
\\ a \\ b \\ c \\ d \\ e \\ f
\end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0
\end{bmatrix}
\end{array}
$$

mark	✓	✓	✓	✓	✓	✓

DFS Tree



$f_{5,1}$

$c_{4,2}$

$d_{3,3}$ $e_{6,4}$ ← pop order

$b_{2,5}$ ↑ push order

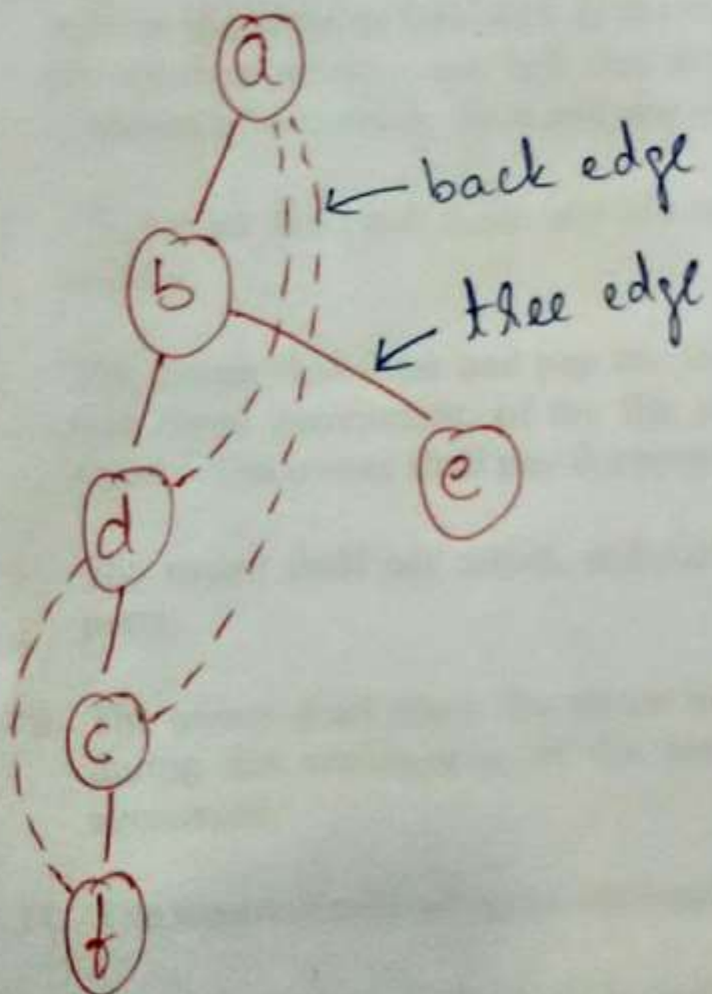
$a_{1,6}$

DFS Stack

mark

✓	✓	✓	✓	✓	✓

DFS Tree



$f_{5,1}$

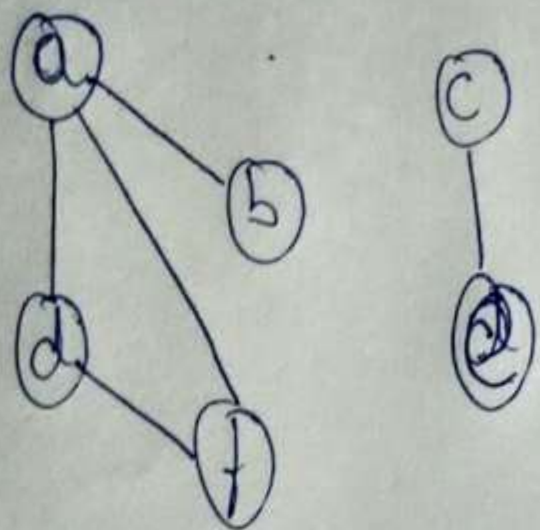
$c_{4,2}$

$d_{3,3}$ $e_{6,4}$ ← pop order

$b_{2,5}$ ↑ push order

$a_{1,6}$

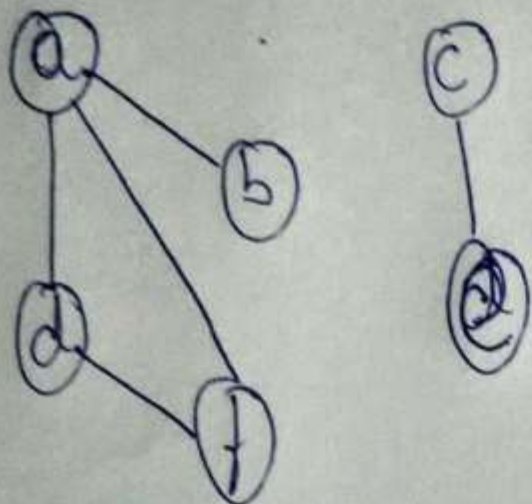
DFS Stack



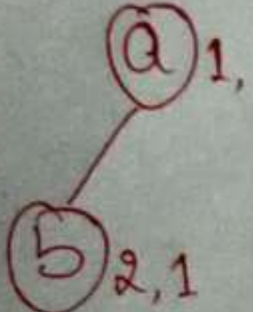
$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 1 & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

a 1,

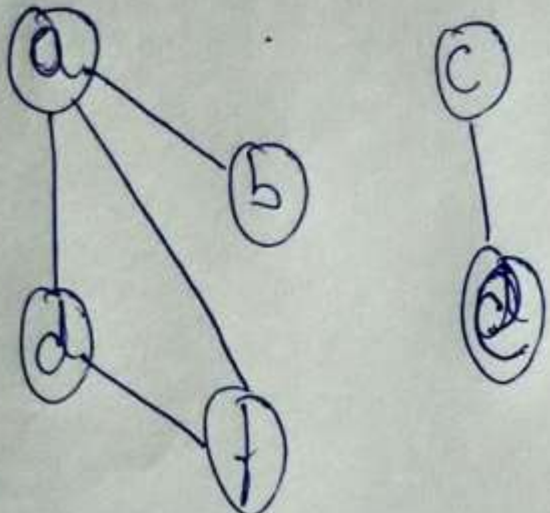
mark						
	1					



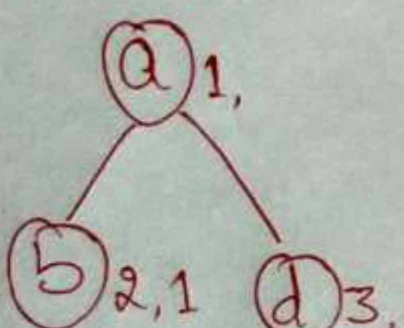
$$
\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
\hline
a & 0 & 1 & 0 & 1 & 0 & 1 \\
b & 1 & 0 & 0 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 0 & 1 \\
e & 0 & 0 & 1 & 0 & 0 & 0 \\
f & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$



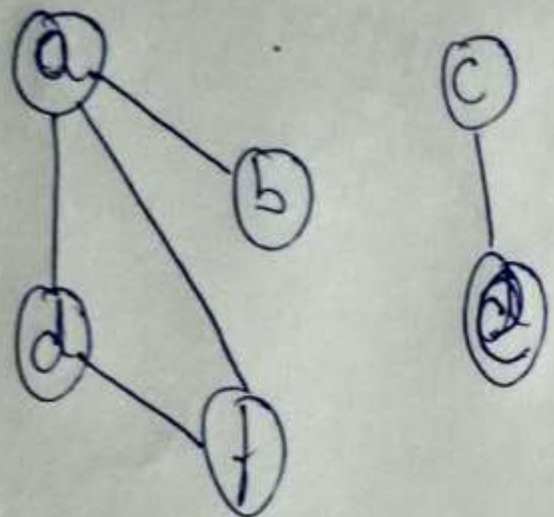
Mark	1	2				



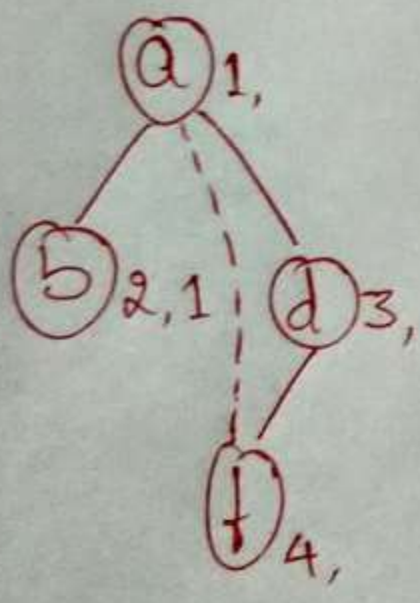
$$
\begin{array}{c c}
 & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} &
\left[\begin{array}{cccccc}
0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$



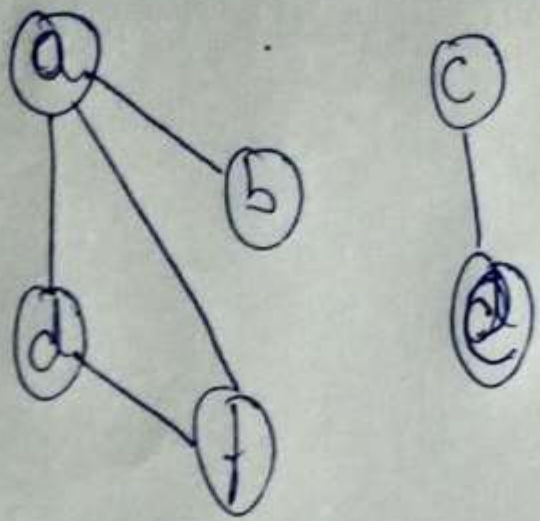
mark	1	2		3		



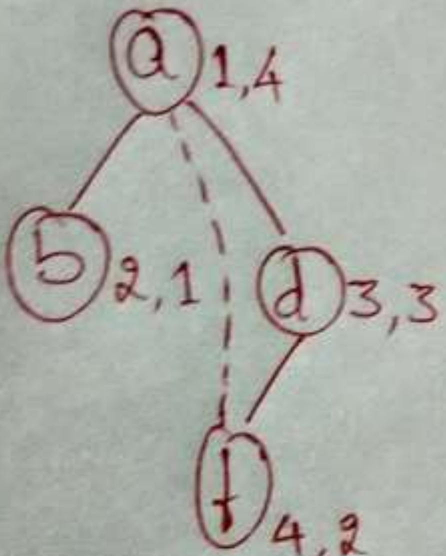
$$
\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
\hline
a & 0 & 1 & 0 & 1 & 0 & 1 \\
b & 1 & 0 & 0 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 0 & 1 \\
e & 0 & 0 & 1 & 0 & 0 & 0 \\
f & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$



mark	1	2		3		4



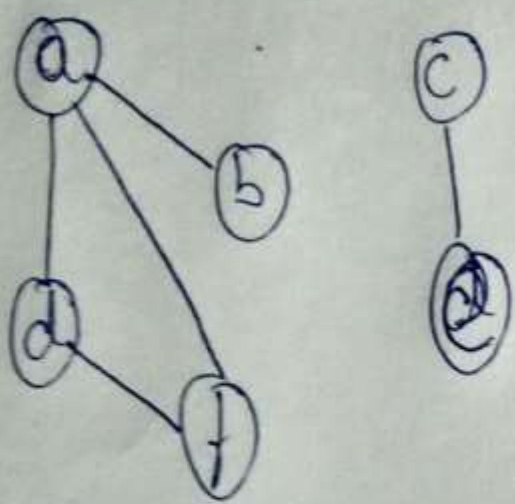
$$
\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
\hline
a & 0 & 1 & 0 & 1 & 0 & 1 \\
b & 1 & 0 & 0 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 0 & 1 \\
e & 0 & 0 & 1 & 0 & 0 & 0 \\
f & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$



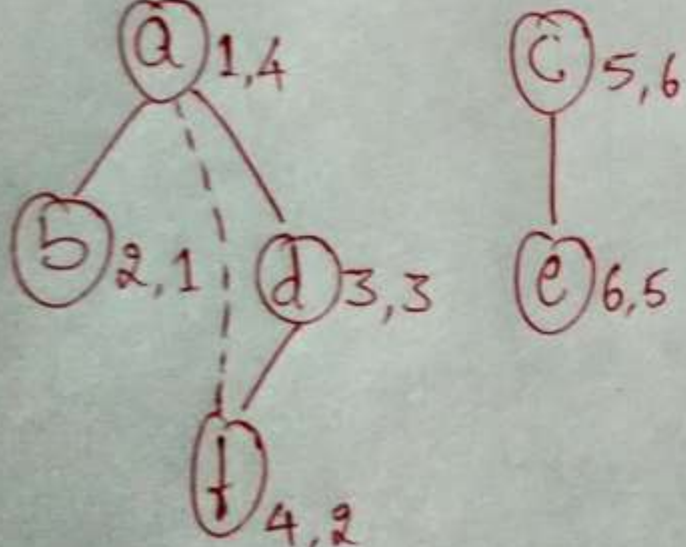
mark	1	2		3		4

```
Algorithm DFS_MAIN( G(V, E) )
    Mark each vertex in v with 0
    for each vertex v in V
        if (v is marked with 0)
            dfs_recurse(v)

Procedure dfs_recurse(v)
    Mark v with 1
    for each vertex w in V adjacent to v
        if (w is marked with 0)
            dfs_recurse(w)
```



$$
\begin{array}{c}
\\
a \\ b \\ c \\ d \\ e \\ f
\end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
\end{array}
$$



Mark	1	2	5	3	6	4

DFS Forest with Stack trace

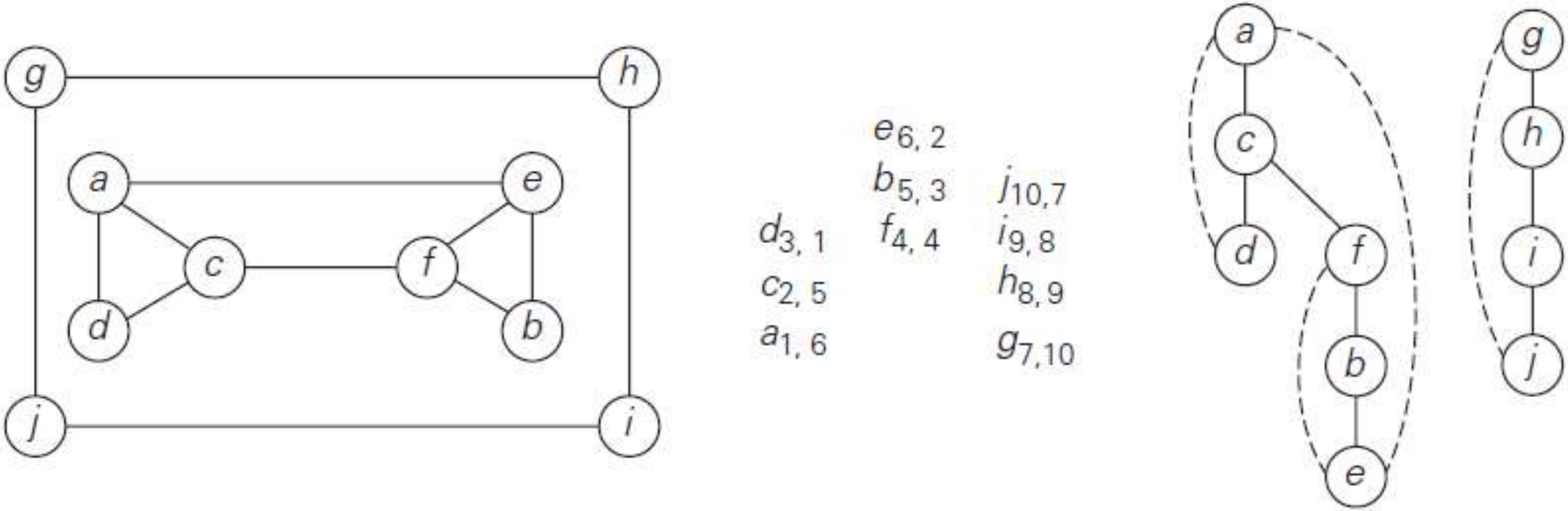
ALGORITHM *DFS(G)*

```
//Implements a depth-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've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 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 ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

```
Algorithm DFS_MAIN( G(V, E) )
    for each vertex v in V
    v.mark ← 0
    counter ← 0
    for each vertex v in V
        if (v is marked with 0)
            dfs_recurse(v)

Procedure dfs_recurse(v)
    counter ← counter + 1
  v.mark ← counter
    for each vertex w in V adjacent to v
        if (w.mark = 0)    dfs_recurse(w)
```



(a) (b) (c)

(a) Graph

(b) Stack of the DFS Traversal

(c) DFS Forest

i. Tree edges

ii. Back edges

Time Efficiency of DFS(G):

Adjacency Matrix:

$\Theta(\ |V|^2\ )$

Adjacency Lists:

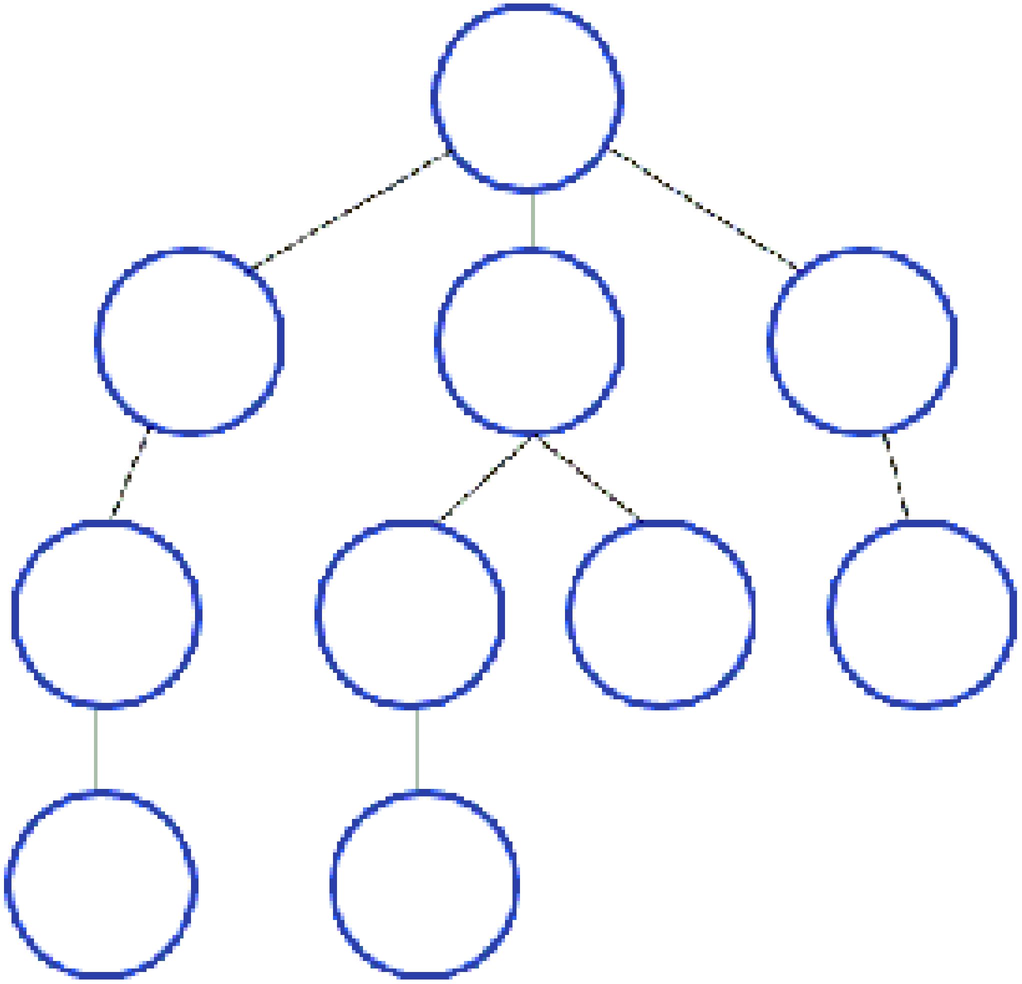
$\Theta(\ |V| + |E|\ )$

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 ← 0
for each vertex *v* in *V* **do**
    if *v* is marked with 0
        dfs(*v*)

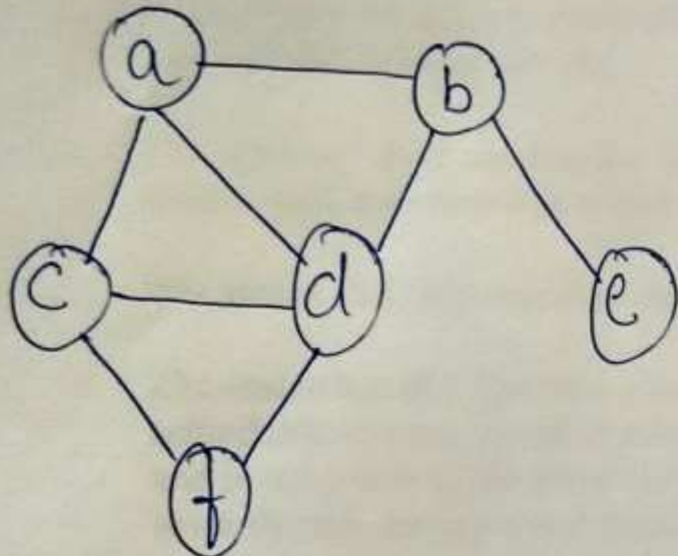
dfs(*v*)
//visits recursively all the unvisited vertices connected to vertex *v* by a path
//and numbers them in the order they are encountered
//via global variable *count*
count ← *count* + 1; mark *v* with *count*
for each vertex *w* in *V* adjacent to *v* **do**
    if *w* is marked with 0
        dfs(*w*)

Breadth First Search (BFS)

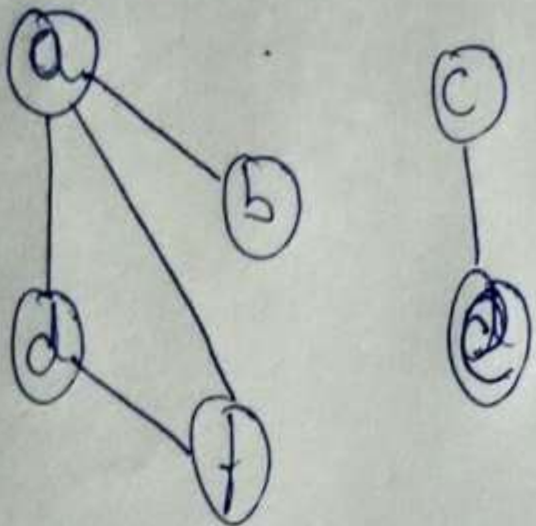


```
Algorithm BFS_main( G(V, E) )
      Mark each vertex in v with 0
      for each vertex v in V
            if(v is marked with 0)
                  bfs_node(v)

Procedure bfs_node(v)
      Mark v with 1
   Insert v into the Queue
      while the Queue is not empty
      v ← remove a vertex from the Queue
      for each vertex w in V adjacent to v
            if(w is marked with 0)
                  Mark w with 1
                        Add w to the Queue
```



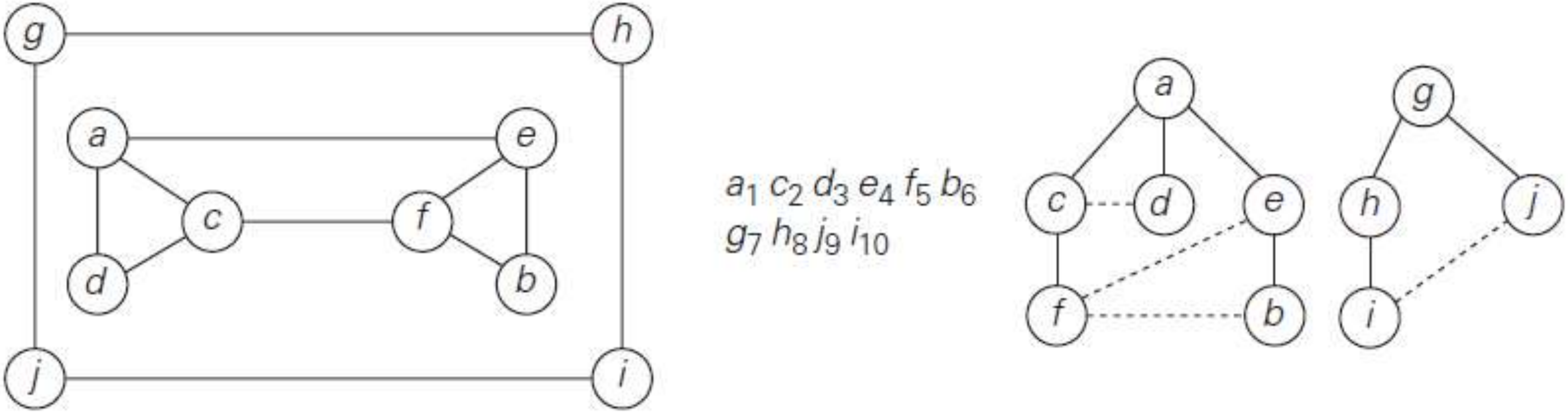
$$
\begin{array}{c}
\\ a \\ b \\ c \\ d \\ e \\ f
\end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\left[\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$



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

a 1,

mark						
	1					



(a) (b) (c)

(a) Graph

(b) Queue of the BFS Traversal

(c) BFS Forest

   i. Tree edges

   ii. Cross edges

Algorithm BFS(G)

```
    for each vertex v in V
    v.mark ← 0
    for each vertex v in V
        if(v.mark = 0)
            v.mark ← 1
        Insert v into the Queue
        while the Queue is not empty
        v ← remove a vertex from the Queue
        for each vertex w in V adjacent to v
            if(w.mark = 0)
                w.mark ← 1
                    Add w to the Queue
```

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 are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 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 numbers them in the order they are visited
//via global variable count
count ← 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 ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

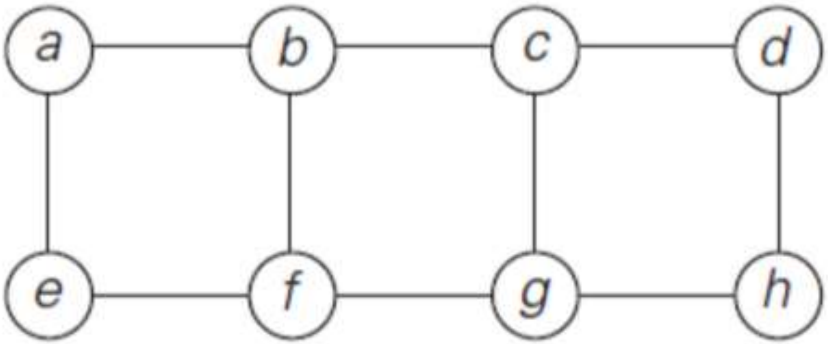
Time Efficiency of BFS(G):

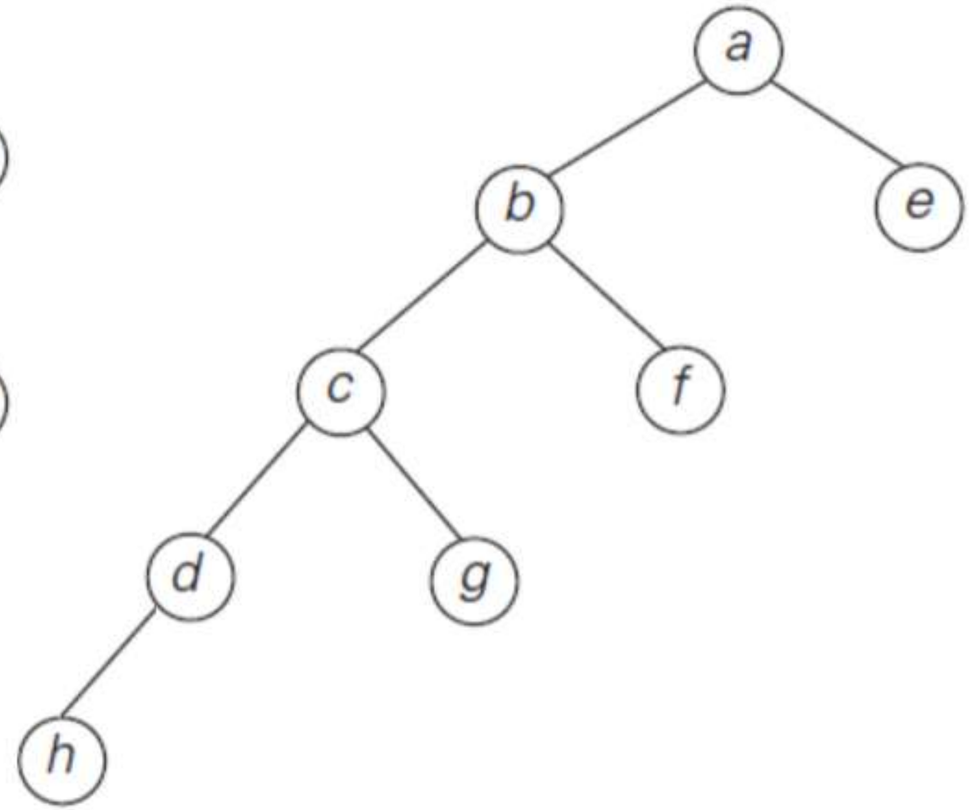
Adjacency Matrix: $\boldsymbol{\Theta(|V|^2)}$

Adjacency Lists: $\boldsymbol{\Theta(|V| + |E|)}$

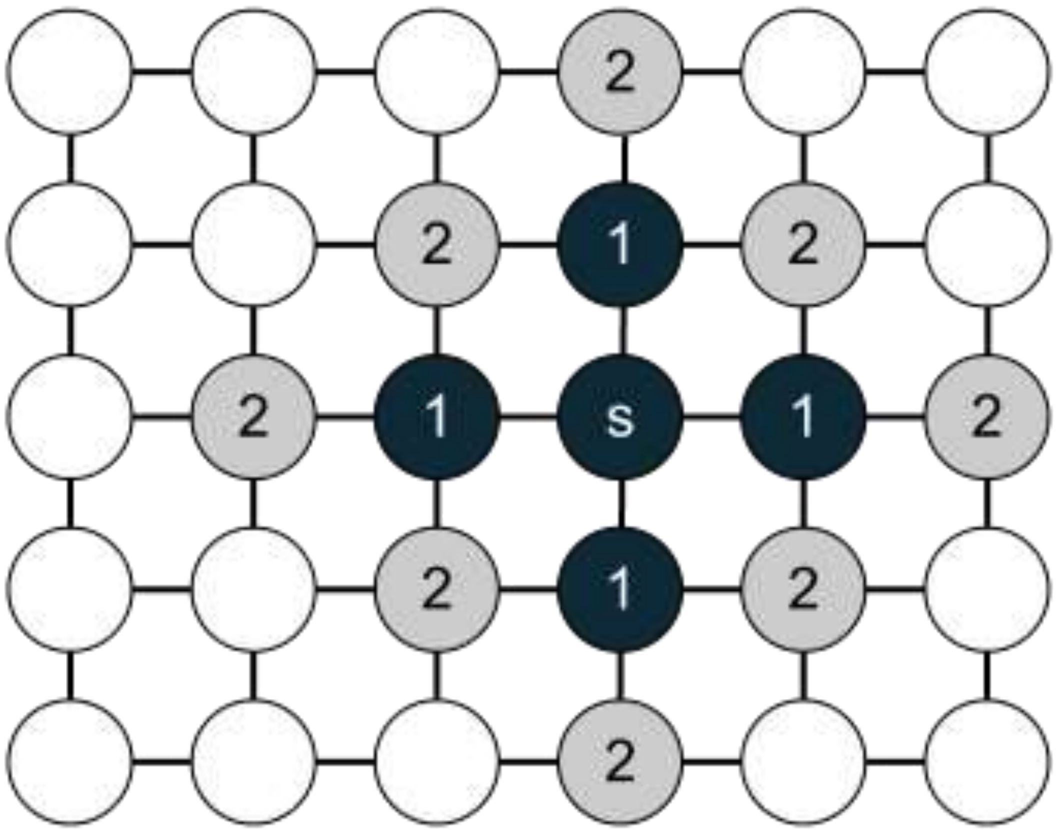
```
Algorithm BFS_main( G(V, E) )
      Mark each vertex in v with 0
      for each vertex v in V
            if(v is marked with 0)
                  bfs_node(v)
Procedure bfs_node(v)
      Mark v with 1
   Insert v into the Queue
      while the Queue is not empty
      v ← remove a vertex from the Queue
      for each vertex w in V adjacent to v
            if(w is marked with 0)
                  Mark w with 1
                        Add w to the Queue
```

BFS-based algorithm for finding a minimum-edge path.





BFS: WHITE, GRAY, BLACK



Paths

The distances between the starting vertex and all the other vertices are the shortest possible!

Problem for the lab on DFS/BFS:

Write a program to find the number of components an undirected graph has. The graph could be represented as an adjacency matrix.

(a) Using DFS traversal

(b) Using BFS traversal

The output is just a positive integer indicating the number of components the given undirected graph has.

Problem for the lab on DFS/BFS: Test-cases:

```
Sample Input 1:
3
0 1 0
1 0 0
0 0 0

Sample Output 1:
2

Sample Input 2:
2
0 0
0 0

Sample Output 2:
2
```

Problem for the lab on DFS/BFS: Test-cases:

```
Sample Input 3:
1
1

Sample Output 3:
1

Sample Input 4:
9
0 1 1 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 1
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0

Sample Output 4:
4
```

Practice problem for the lab on DFS/BFS:

Simulate a **maze search** using **DFS** technique. Represent a 2-dimensional maze by grid of nodes, where a node in the grid can potentially connect to four other nodes in the grid. Two nodes have an edge between them if they are adjacent in the grid and there is no obstacle in between. Print the index of the nodes (i, j) visited during the maze search. The search ends when all reachable nodes from (0, 0) are visited.

Practice problem for the lab on DFS/BFS:

...

Input: **'r'** rows and **'c'** columns in the grid. It is followed by **r*c** binary strings of 4 bits (like 1001) indicating the connection to its right, bottom, left and top adjacent nodes, respectively.

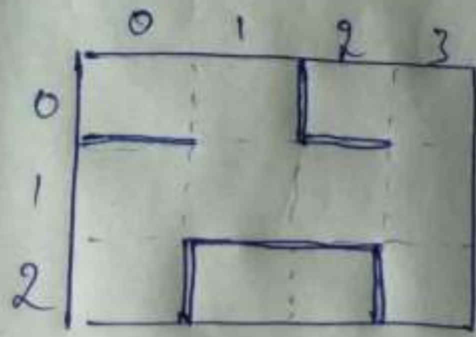
Output: Series of nodes with their locations indices (like (4,0) indicating 4th row, 0th column) in the order of their visit during the maze search. The ones visited during backtracking also to be printed.

Practice problem for the lab on DFS/BFS:

...

Sample Input/ Output:

Maze Search



Sample Input/output

Input: 3 4

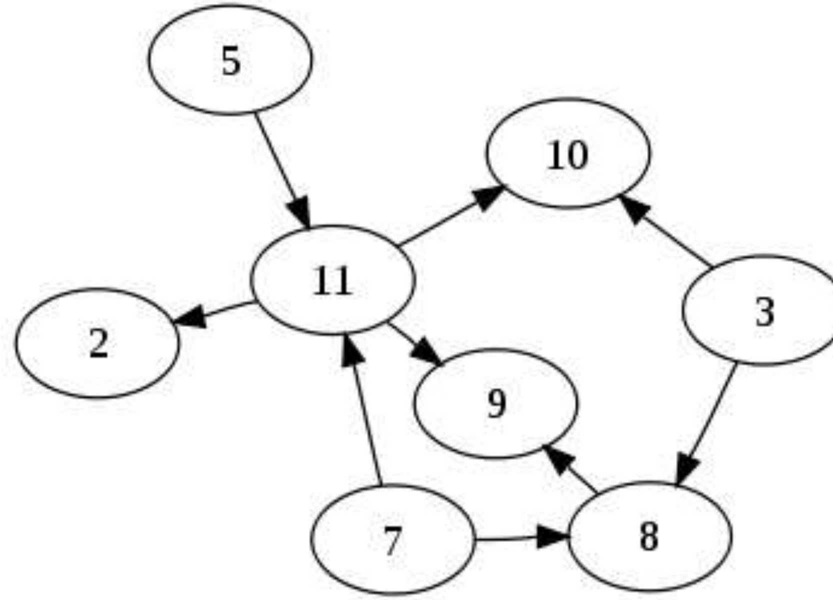
1000 0110 1000 0110
1100 1011 1010 0111
0001 1000 0010 0001

Output:

(0,0) (0,1) (1,1) (1,2) (1,3)
(2,3) (1,3) (0,3) (0,2) (0,3)
(1,3) (1,2) (1,1) (1,0) (2,0)
(1,0) (1,0) (0,1) (0,0)

Directed cycle:

It is a cycle or circuit in a digraph.



DAG (**directed acyclic graph**)**:**

A digraph having no directed cycle.

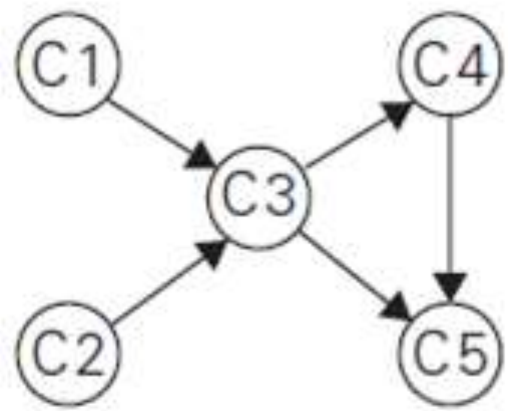
Topological Sorting:

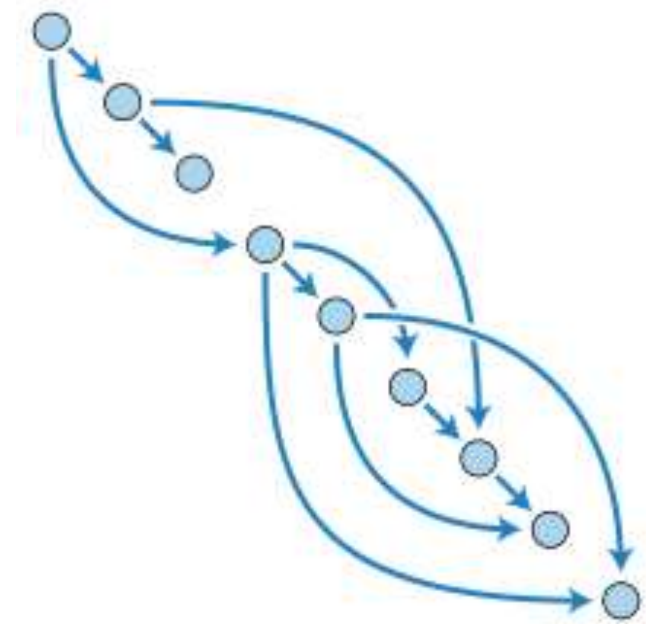
(aka Toposort, Topological Ordering)

What do you know about **Topological Sorting** from the course in Discrete Math?

Topological Sorting: is listing vertices of a directed graph 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.

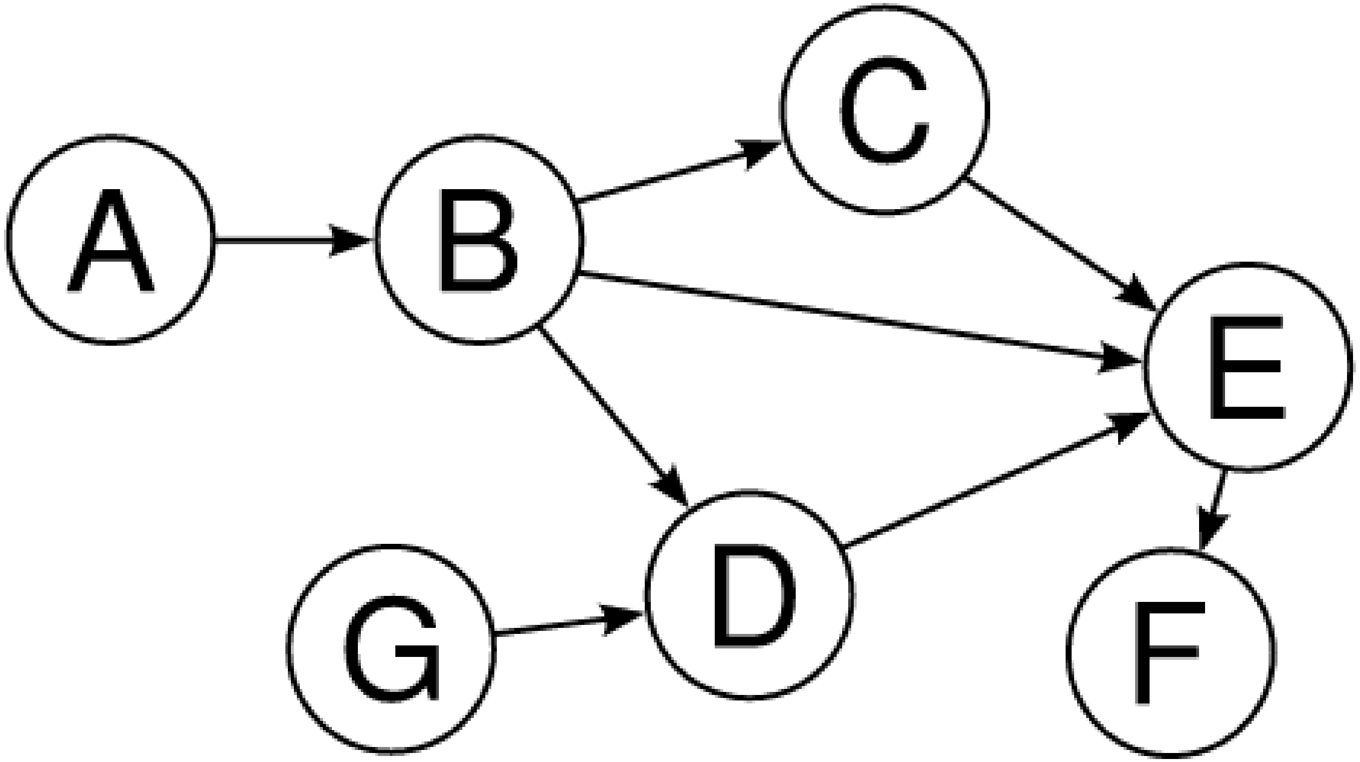
A digraph has a topological sorting iff it is a **dag**.



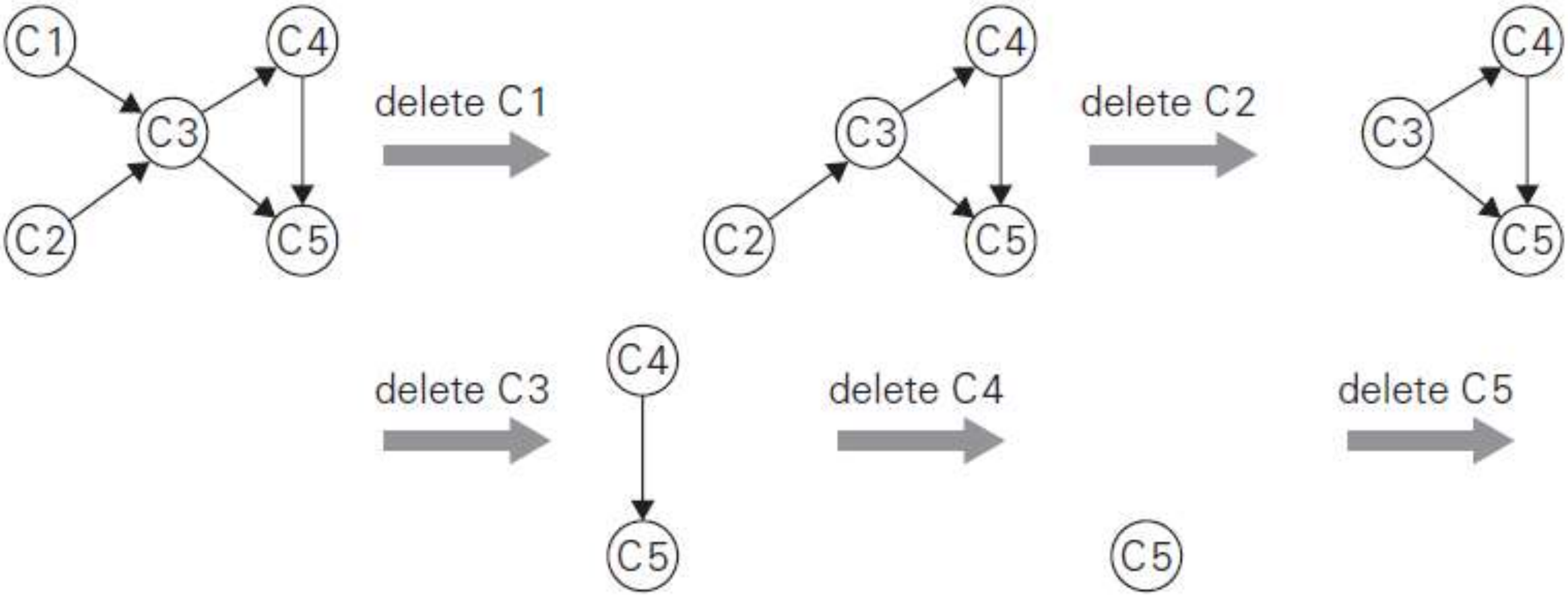


Finding a **Topological Sorting** of the vertices of a dag:

- **Source-removal** algorithm
- **DFS-based** algorithm



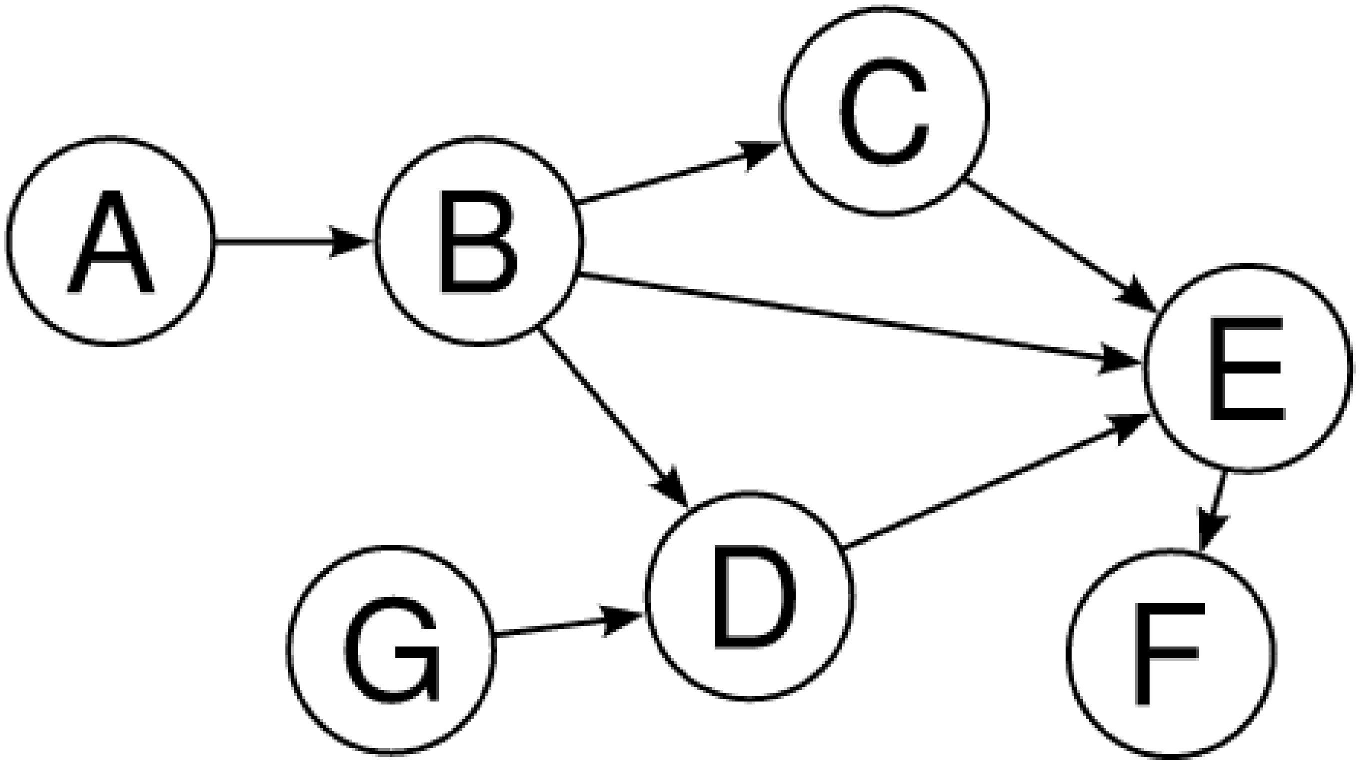
Source-removal algorithm for finding **Topological Sorting**



The solution obtained is C1, C2, C3, C4, C5

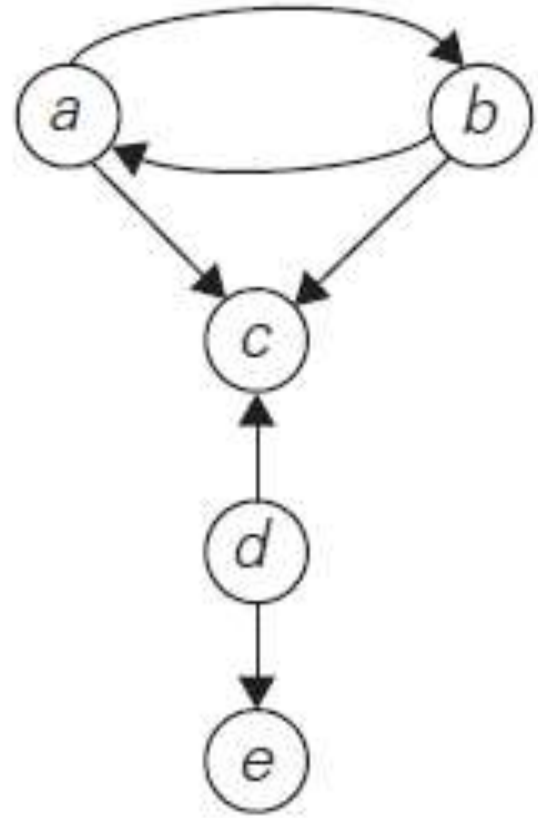
```
Algorithm SourceRemoval_Toposort(V, E)
L ← Empty list that will contain the sorted vertices
S ← Set of all vertices with no incoming edges
while S is non-empty do
    remove a vertex v from S
    add v to tail of L
    for each vertex m with an edge e from v to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (not a DAG)
else return L (a topologically sorted order)
```

Find a Topological Sort of the following **dag** using **Source-removal** algorithm



Draw a DFS forest for

the given directed graph.



(a) DiGraph

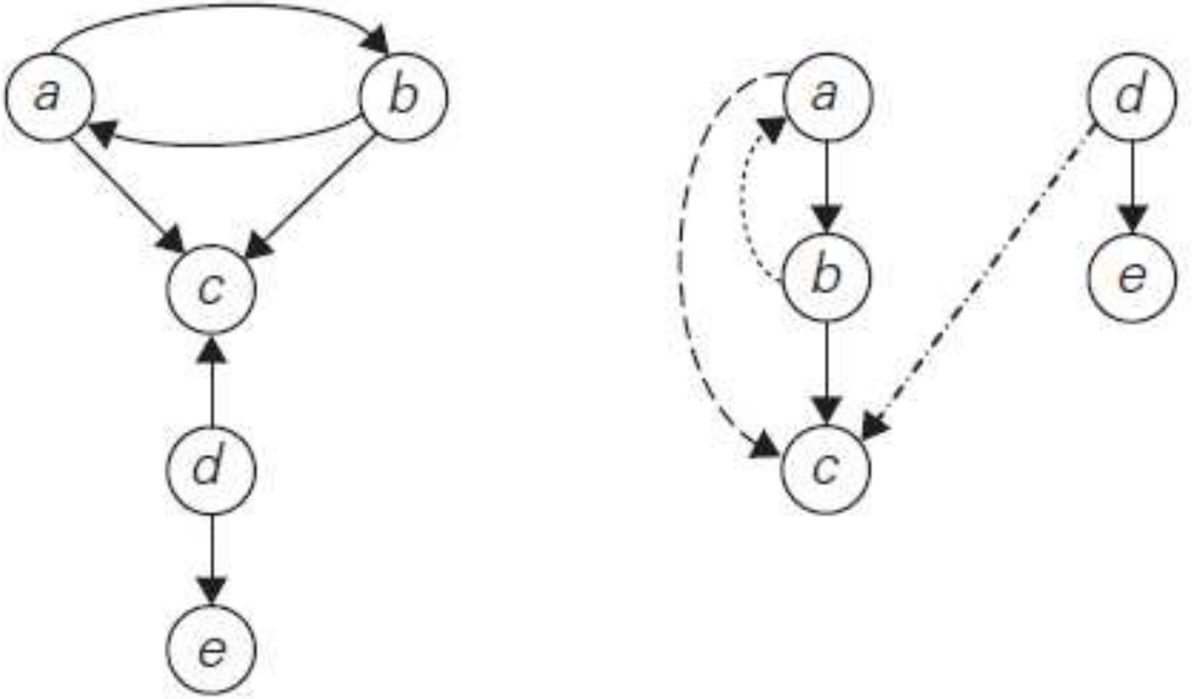
(b) DFS Forest

i. Tree edges

ii. Back edges

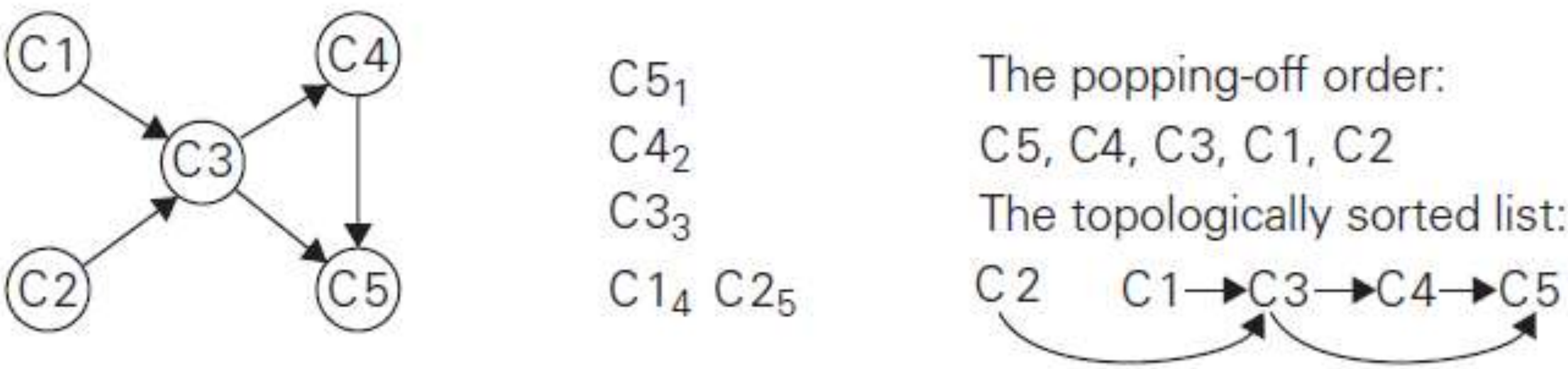
iii. Forward edges

iv. Cross edges



(a) (b)

DFS-based algorithm for finding **Topological Sorting**



(a) (b) (c)

Algorithm DFS_Toposort(V, E)

L ← Empty list that will contain the sorted vertices

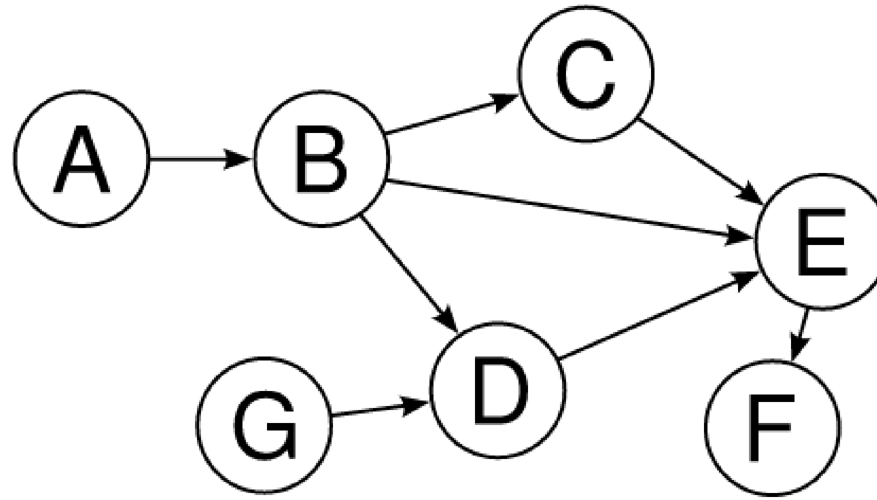
for each vertex v in V

    v.mark ← 0

for each vertex v in V **do**

    if(v.mark = 0) visit(v, L)

return L



Procedure visit(vertex v, List L)

    v.mark ← 1

    for each vertex m with an edge (v, m) **do**

        if(m.mark = 0) visit(m)

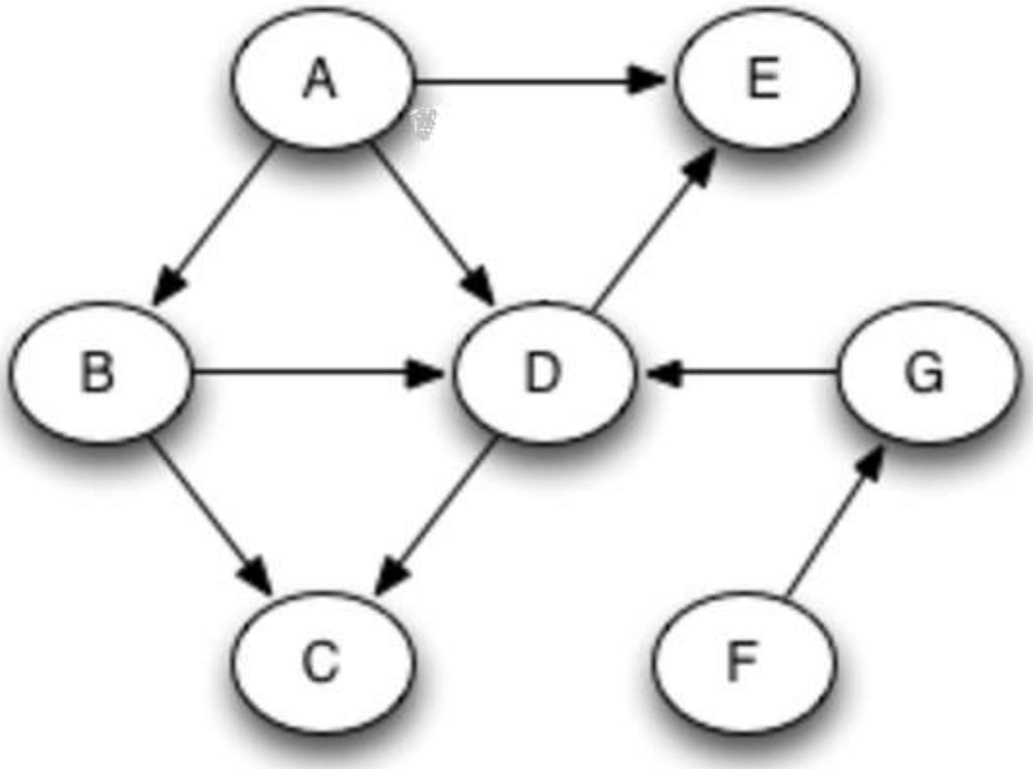
    add v at *head* of L

```
Algorithm DFS_Toposort(V, E)
L ← Empty list that will contain the sorted vertices
for each vertex v in V
     mark(v) ← 0, instack(v) ← 0
for each vertex v do
    if(mark(v) = 0) visit(v, L)
return L

Procedure visit(vertex v, List L)
    mark(v) ← 1, instack(v) ← 1
    for each vertex m with an edge (v, m) do
        if(mark(v) = 0) visit(m)
        else if(instack(m) = 1) return ERROR_DAG
    add v at head of L
    instack(v) ← 0
```

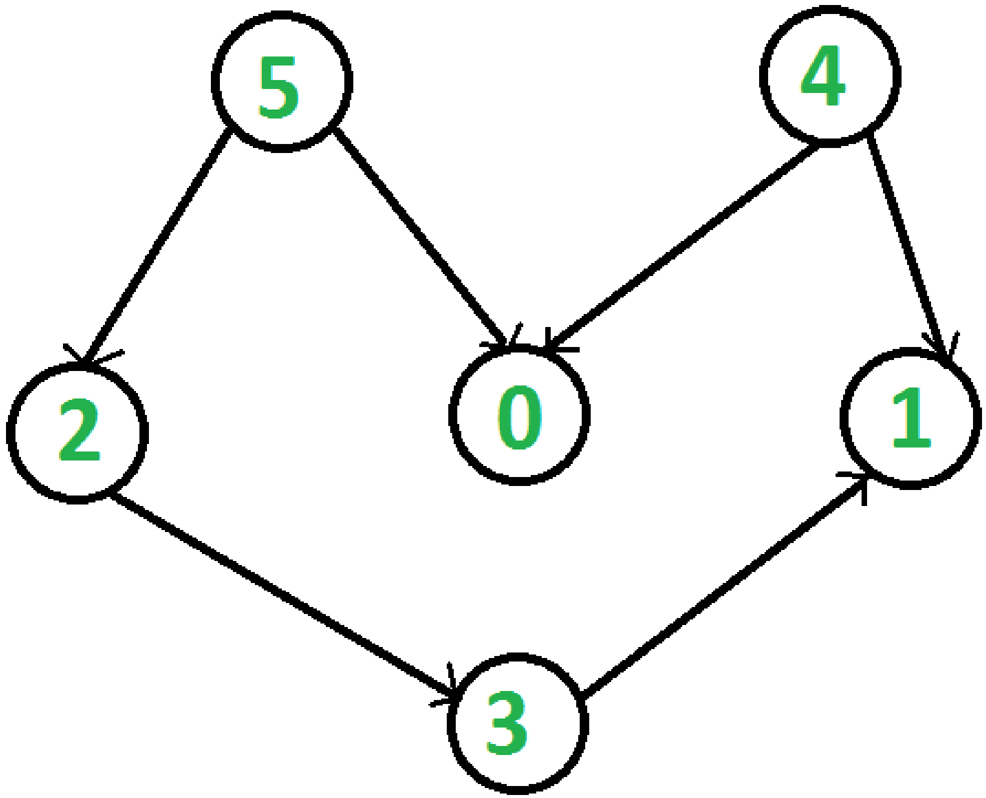
Topological Sorting of the vertices using

- **Source-removal** algorithm
- **DFS-based** algorithm



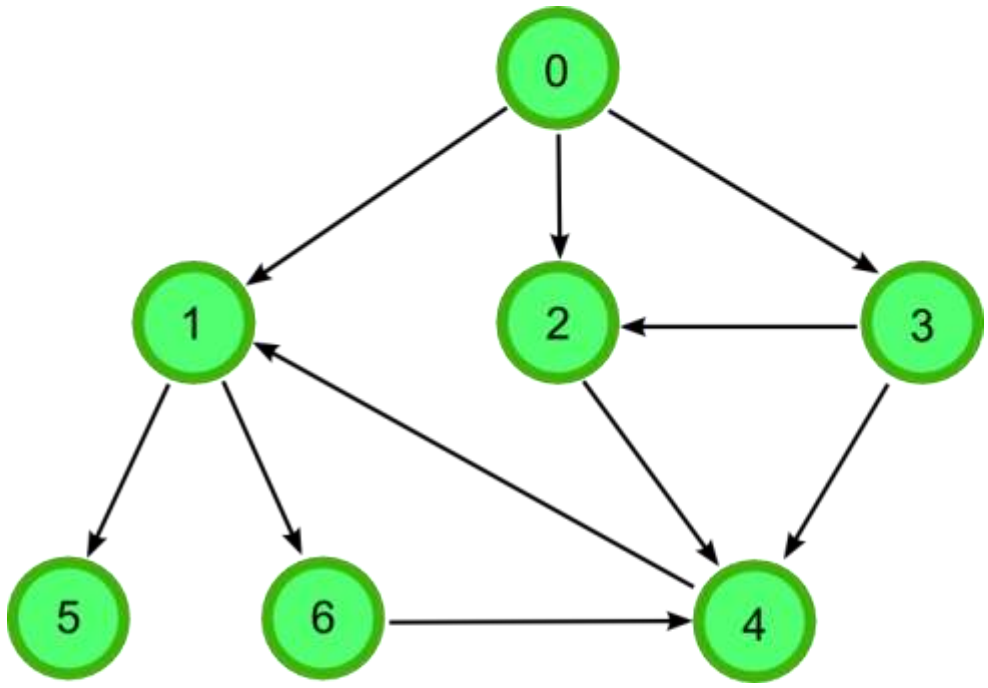
Topological Sorting of the vertices using

- **Source-removal** algorithm
- **DFS-based** algorithm



Topological Sorting of the vertices using

- **Source-removal** algorithm
- **DFS-based** algorithm



Generating Permutations:

- Lexicographic Order

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
    let i be its largest index such that $a_i < a_{i+1}$ $//a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index j such that $a_i < a_j$ $//j \geq i + 1$ since $a_i < a_{i+1}$
    swap a_i with a_j $//a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list

- Decrease-and-Conquer

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element 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:

123

132

213

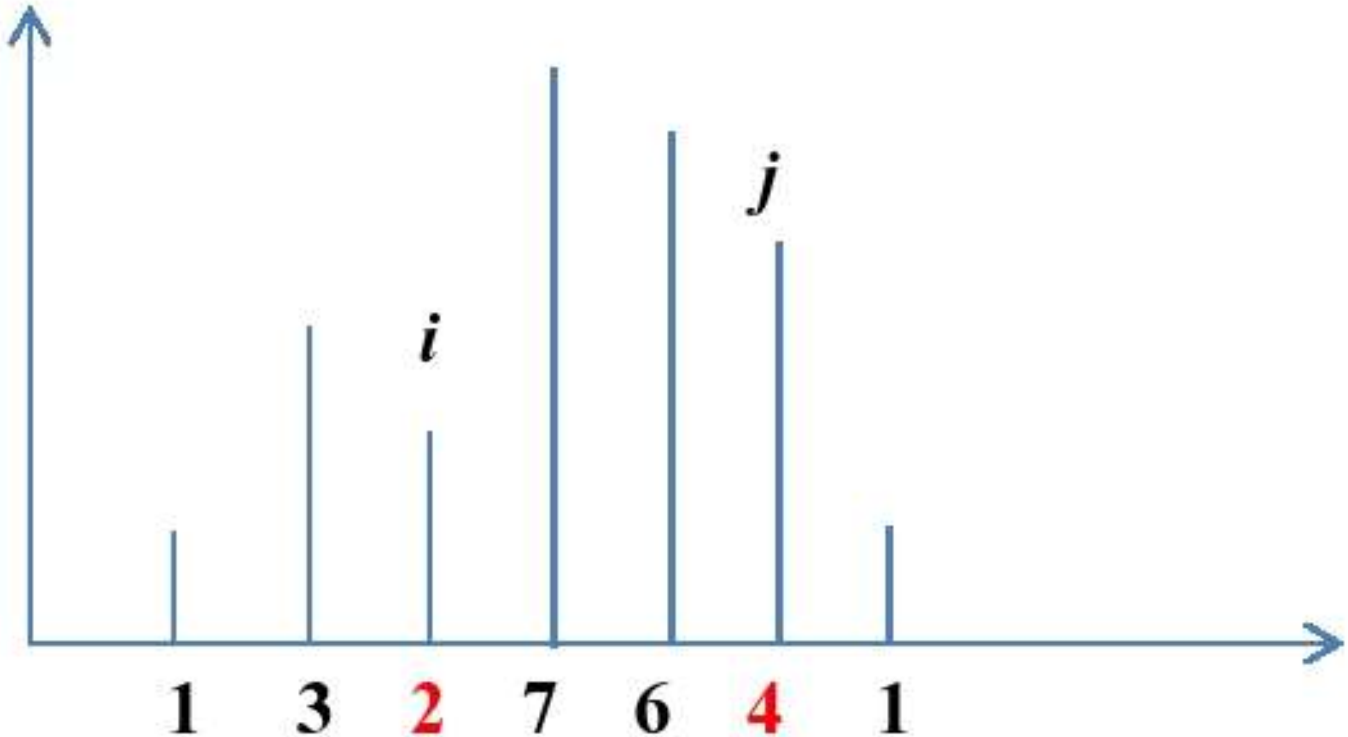
231

312

321

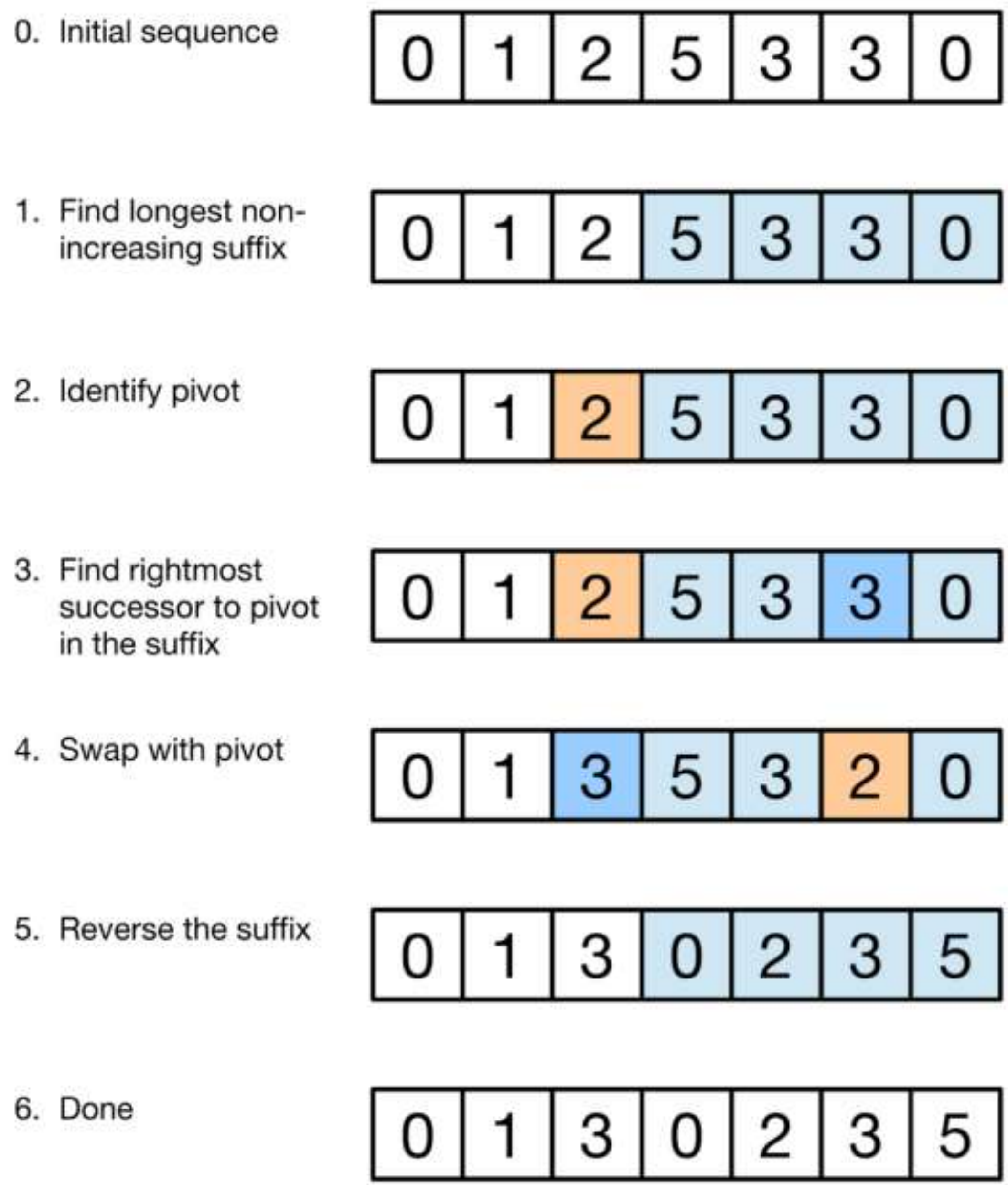
- Lexicographic order
  - the order in which they would be listed in a dictionary if the digits were interpreted as letters/characters.

- Decrease-and-Conquer
  - Solve it for input size of **(n-1)** and hence for **n**.



swap 1 3 4 7 6 2 1

sort 1 3 4 1 2 6 7



0. Initial sequence
0 1 2 5 3 3 0
1. Find longest non-increasing suffix
0 1 2 5 3 3 0
2. Identify pivot
0 1 2 5 3 3 0
3. Find rightmost successor to pivot in the suffix
0 1 2 5 3 3 0
4. Swap with pivot
0 1 3 5 3 2 0
5. Reverse the suffix
0 1 3 0 2 3 5
6. Done
0 1 3 0 2 3 5

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
    let i be its largest index such that $a_i < a_{i+1}$ //$a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index j such that $a_i < a_j$ //$j \geq i + 1$ since $a_i < a_{i+1}$
    swap a_i with a_j //$a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list

Generating Permutations by **Decrease-and-Conquer**

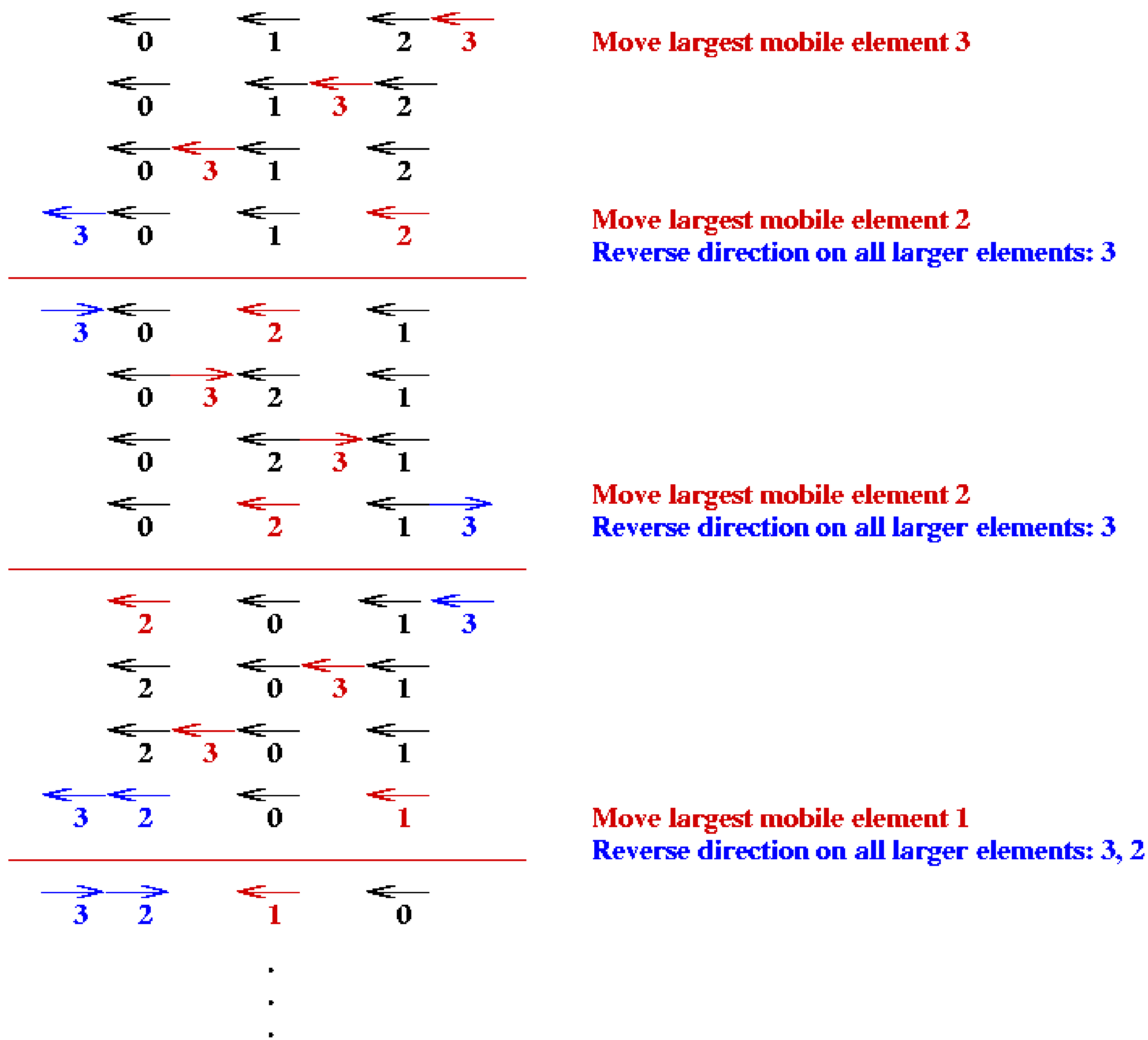
Solve it for input size of **(n-1)** and hence for **n**.

Step 1: There is only one permutation for 1 symbol

Step 2: Assume we know how to generate permutations for (n-1) symbols.

Step 3: Extend it to generate permutations for n symbols.

Johnson-Trotter algorithm to generate permutations



Johnson-Trotter algorithm
to generate permutations

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{3}$	$\overleftarrow{1}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{3}$	$\overleftarrow{4}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}}$
$\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{3}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overleftarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2}\,\overrightarrow{4}$
$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{2}$	$\overleftarrow{3}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{1}\,\overleftarrow{2}$	$\overleftarrow{4}\,\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}}$
$\overrightarrow{\mathbf{4}}\,\overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}$	$\overrightarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1}$	$\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}$	$\overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{4}$
$\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{1}$	$\overleftarrow{2}\,\overleftarrow{\mathbf{4}}\,\overrightarrow{3}\,\overleftarrow{1}$	$\overleftarrow{4}\,\overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1}$
$\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}\,\overrightarrow{4}$

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element 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 Subsets:

Knapsack problem needed to find the most valuable subset of items that fits a knapsack of a given capacity.

Powerset: set of all subsets of a set. Set A={1, 2, ..., n} has $\mathbf{2^n}$ subsets.

Generate all subsets of the set A={1, 2, ..., n}.

Any **decrease-by-one** idea?
# of subsets of { } = 2^0 = 1, which is **{ }** itself
Suppose, we know how to generate all subsets of {1,2,...,n-1}
Now, how can we generate all subsets of {1,2,...,n} ?

Generating Subsets:

All subsets of {1,2,...,n-1}: 2^{n-1} such subsets

All subsets of {1,2,...,n}:
2^{n-1} subsets of {1,2,...,n-1} and
another 2^{n-1} subsets of {1,2,...,n-1} having **‘n’** with them.

That adds up to all 2^n subsets of {1,2,...,n}

0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Alternate way of Generating Subsets:

Knowing the binary nature of either having **n**th element or not, any idea involving binary numbers itself?

One-to-one correspondence between all 2^n bit strings $b_1b_2...b_n$ and 2^n subsets of $\{a_1, a_2, \ldots, a_n\}$.

Each bit string $b_1b_2...b_n$ could correspond to a subset.
In a bit string $b_1b_2...b_n$, depending on whether b_i is 1 or 0, a_i is in the subset or not in the subset.

000	001	010	011	100	101	110	111
$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in **Squashed order:**

Squashed order: any subset involving a_j can be listed only after all the subsets involving a_1, a_2, …, a_{j-1}

Both of the previous methods does generate subsets in squashed order.

0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in **Squashed order:**

Squashed order: any subset involving a_j can be listed only after all the subsets involving a_1, a_2, …, a_{j-1}

Can we do it with minimal change in bit-string (actually, just one-bit change to get the next bit string)? This would mean, to get a new subset, just change one item (remove one item or add one item).

Binary reflected gray code:

000   001   011   010   110   111   101   100

Decrease-by-a-Constant-Factor Algorithms:

Finding a^n

$$a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}$$

- $a^n = (a^{n/2})^2$ when n is even

  $a^n = a*(a^{(n-1)/2})^2$ when n is odd and

  $a^1 = a$, $a^0 = 1$

Binary Search:

$$\begin{array}{ccc} & K & \\ & \updownarrow & \\ \underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} & A[m] & \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}} \end{array}$$

Decrease-by-a-Constant-Factor Algorithms:

```
Algorithm BinarySearchRec(A[0..n-1], K)
      if(n <= 0)
            return -1
      m = ⌊n/2⌋
      if(k = A[m])
            return m
      if(k < A[m])
            return BinarySearchRec(A[0..m-1], K)
      else
            return BinarySearchRec(A[m+1..n-1],
K)
```

Multiplication à la Russe:
(aka Russian peasant method)

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	$+(130+1040)=3250$

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

$$1 \cdot m = m$$

A puzzle circulated in WhatsApp groups

There are 8 coins. Out of which one is fake.
The fake coin is lighter in weight than others.
You have a common balance to weigh the coins.

How many iterations of weighing are required,
to find the fake coin?

Fake-Coin Problem: There are ***n*** coins, which appear identical except that one of them is fake. The fake coin is lighter than the genuine ones. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much). Design an efficient algorithm for detecting the fake coin. Objective is to minimize the number of iterations of weighing.

Decrease-by-a-Constant-Factor Algorithms:

Fake-Coin Problem:

1. Decrease-by-a-factor of 2 algorithm
   For 8 coins, it takes 3 iterations.
   Ceil($\log_2$ n) iterations of weighing
   Ceil($\log_2$ 1 trillion) = 40

1. Decrease-by-a-factor of 3 algorithm
   For 8 coins, it takes 2 iterations.
   Ceil($\log_3$ n) iterations of weighing
   Ceil($\log_3$ 1 trillion) = 26

A puzzle circulated in WhatsApp groups

10 people standing in a circle in an order from 1 to 10.

No. 1 has a sword.

He kills the next person (i.e. no. 2) and gives sword to the following person (i.e. no. 3). Every time the person holding the sword kills the next alive person in the circle and gives the sword to the following alive person. It continues till only one person survives. Which one survives in the end? The problem is to determine the survivor’s number.

A puzzle circulated in WhatsApp groups

Answer for the previous one is 5. That is, 5th person survives. What if there are 100 people in the circle to start with.

100 people standing in a circle in an order from 1 to 100.

No. 1 has a sword.

He kills the next person (i.e. no. 2) and gives sword to the following person (i.e. no. 3). Every time the person holding the sword kills the next alive person in the circle and gives the sword to the following alive person. It continues till only one person survives. Which one survives in the end? The problem is to determine the survivor’s number J(n).

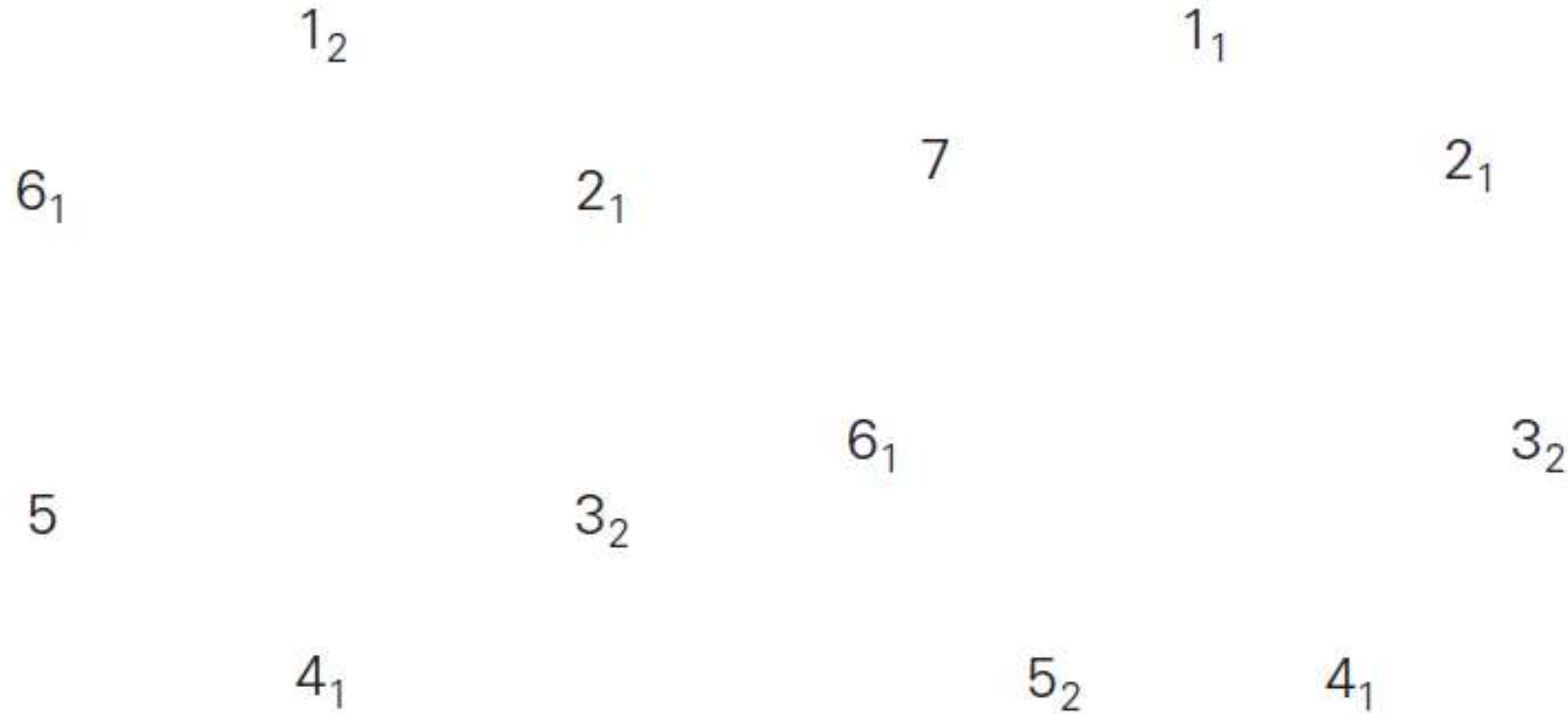
Josephus Problem:

Let n people be numbered from 1 to n stand in a circle. Starting the count from 1, we eliminate every second person until only one survivor is left. The problem is to determine the survivor’s number J(n).

- A group of m soldiers are surrounded by the enemy and there is only a single horse for escape.
- The soldiers determine a pact to see who will escape and summon help.
- The form a circle and pick a number n which is between 1 and m.
- One of their names is also selected at random.

Josephus Problem:

The problem is to determine the survivor’s number J(n).



(a) (b)

Josephus Problem:

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

Josephus Problem:
The problem is to determine the survivor's number J(n).

$$J(2k) = 2J(k) - 1$$

$$J(2k+1) = 2J(k) + 1$$

1_2
6_1 2_1
5 3_2
4_1
(a)

1_1
7 2_1
6_1 3_2
5_2 4_1
(b)

$J(n)$ can be obtained by a 1-bit cyclic shift left of n itself!
$J(6) = J(110_2) = 101_2 = 5$ and $J(7) = J(111_2) = 111_2 = 7$.

Take just a step at a time and reduce your problem size!

</ Decrease-n-Conquer >