Design and Analysis of Algorithms (UE14CS251)

Decrease-and-Conquer

Mr. Channa Bankapur
channabankapur {@pes.edu, @gmail.com}

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

```
Algorithm Sum(A[0..n-1])
//$Sum_{0..n-1}$ = 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
```

$T(n) = n$

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

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

```
Algorithm Sum(A[0..n-1])
//$Sum_{0..n-1} = Sum_{0..n/2} + Sum_{(n/2)+1..n-1}$
//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..$\lfloor (n-1)/2 \rfloor$]) +
            Sum(A[$\lfloor (n-1)/2 \rfloor$+1..n-1])
```

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

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

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

```
Algorithm Sum(A[0..n-1])
//$Sum_{0..n-1} = Sum_{0..n-2} + A[n-1]$
//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]
```

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

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

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

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

    $\in \Theta(n)$

Finding a^n using a **Brute Force** approach.

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

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

Finding a^n using a **Divide-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$
//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$
//Output: $a^n$
    if (n = 0) return 1
    if (n = 1) return a
    return Power(a, $\lfloor n/2 \rfloor$) * Power(a, $\lceil n/2 \rceil$)
```

$T(n) = 2T(n/2) + 1 \in \Theta(n)$

Finding a^n using a **Decrease-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds $a^n = a^{n-1} * a$
//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$
//Output: $a^n$
    if (n = 0) return 1
    return Power(a, n-1) * a
```

$T(n) = T(n-1) + 1 \in \Theta(n)$

This approach is **Decrease-by-a-constant-and-Conquer**

Can we solve it by **Decrease-by-a-constant-factor-and-Conquer?**

Finding a^n using a **Decrease-by-a-constant-factor-and-Conquer** approach.

```
Algorithm Power(a, n)
//Finds $a^n = (a^{\lfloor n/2 \rfloor})^2 * a^{n \bmod 2}$
//Input: $a \in \mathbf{R}$ and $n \in \mathbf{I}^+$
//Output: $a^n$
    if (n = 0) return 1
    p ← Power(a, ⌊n/2⌋)
    p ← p * p
    if (n is odd) p ← p * a
    return p
```

$T(n) = T(n/2) + 2 \in \mathbf{\Theta(\log n)}$

Finding **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)**
  - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$
- Decrease-by-a-constant-and-Conquer in **Θ(n)**
  - $a^n = a^{n-1} * a$
- Decrease-by-a-constant-factor-and-Conquer in **Θ(log 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
    $a^1 = a$, $a^0 = 1$

Decrease-and-Conquer:

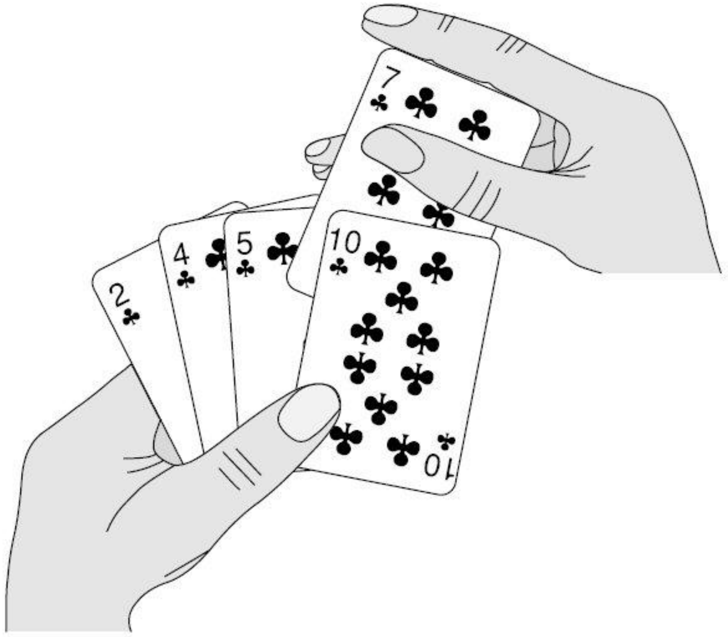
1. Reduce problem instance into a smaller instance of the same problem.
2. Solve the smaller instance.
3. Extend the solution of the smaller instance to obtain solution to the original instance.

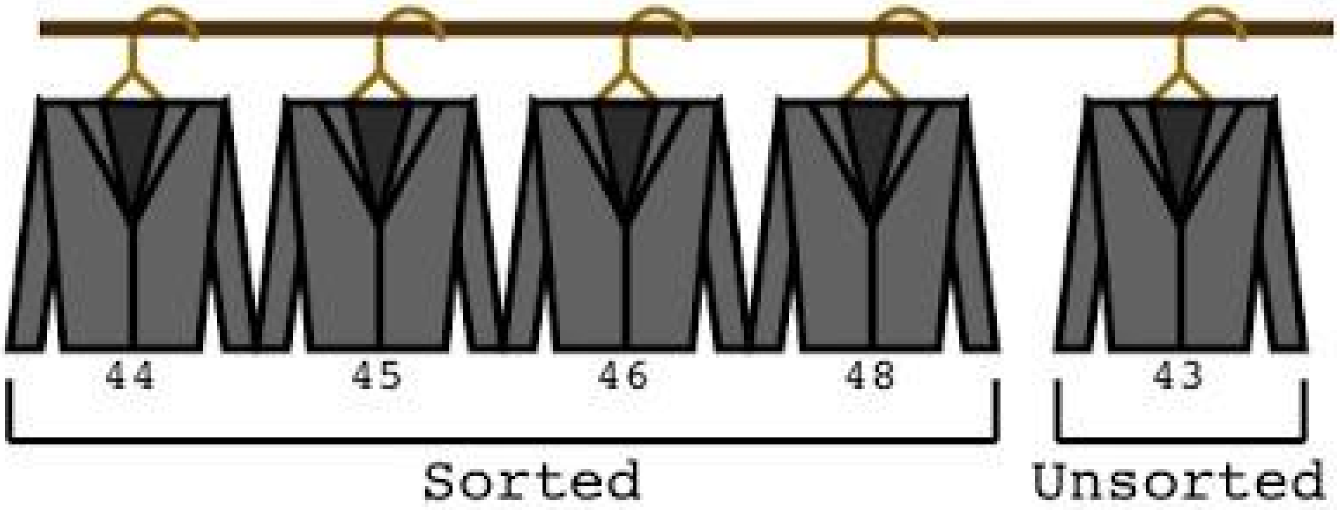
Also referred to as ***inductive*** or ***incremental*** approach.

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:

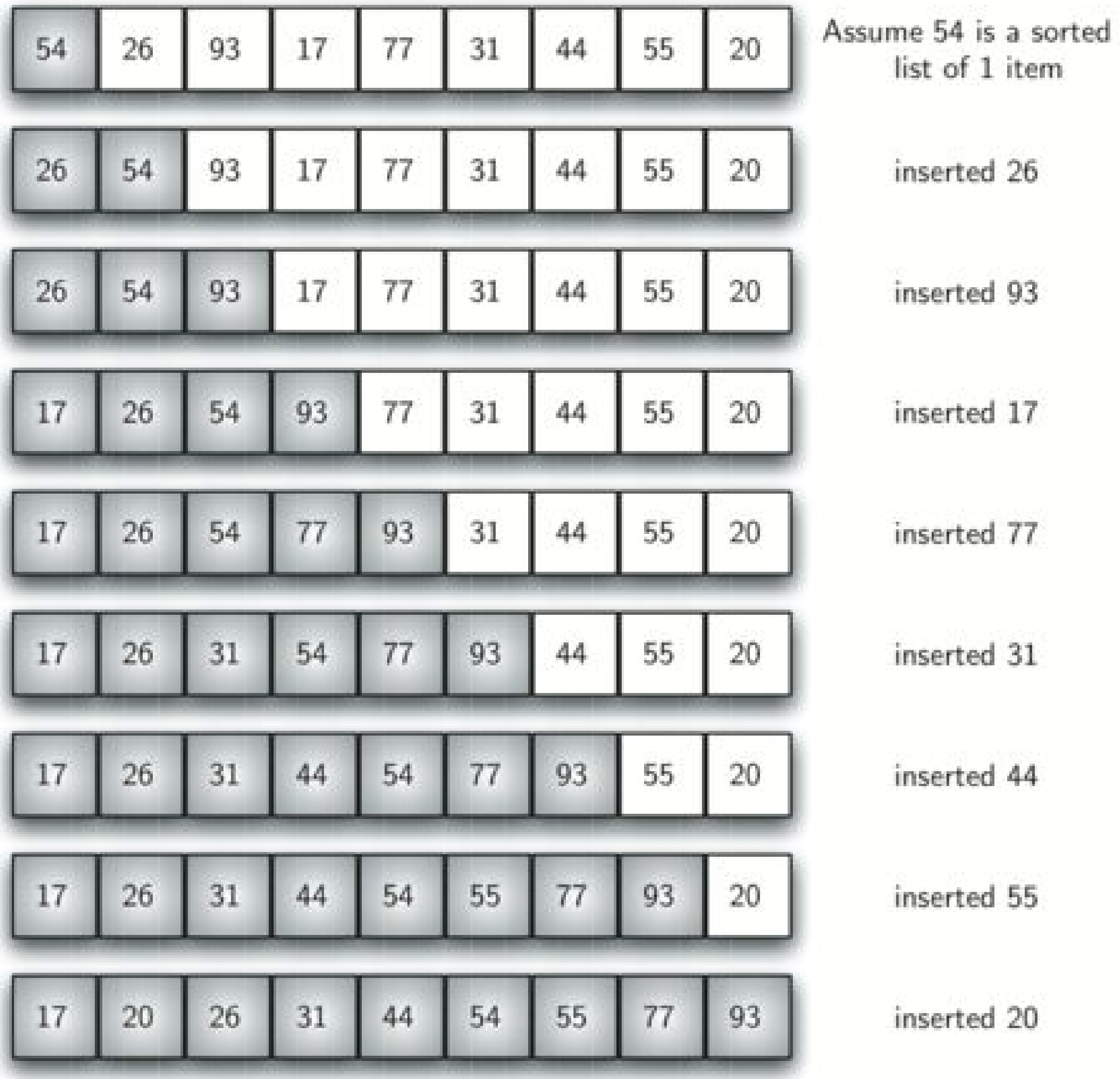


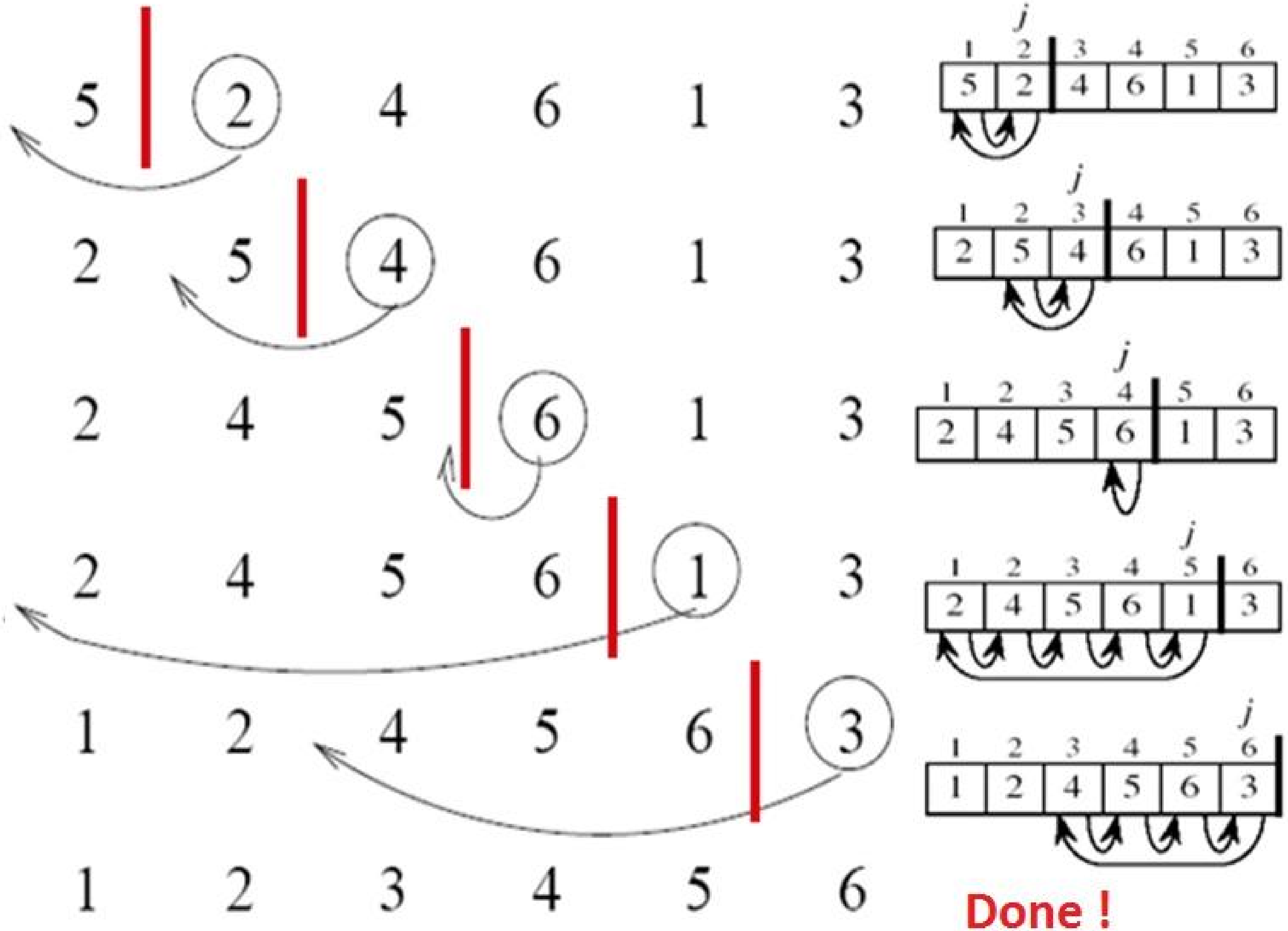


Insertion Sort: To sort an array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2].

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 | 4   1   8   5   5
4   6 | 1   8   5   5
1   4   6 | 8   5   5
1   4   6   8 | 5   5
1   4   5   6   8 | 5
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



Done !

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

```
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 $j \ge 0$ and $A[j] > v$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        $A[j+1] \leftarrow v$
```

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 efficiency ?

Stable sorting ?

Time efficiency

- $C_{worst}(n) = n (n - 1) / 2 \in \Theta(n^2)$
- $C_{best}(n) = n - 1 \in \Theta(n)$
- $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$
- Fast on nearly sorted arrays

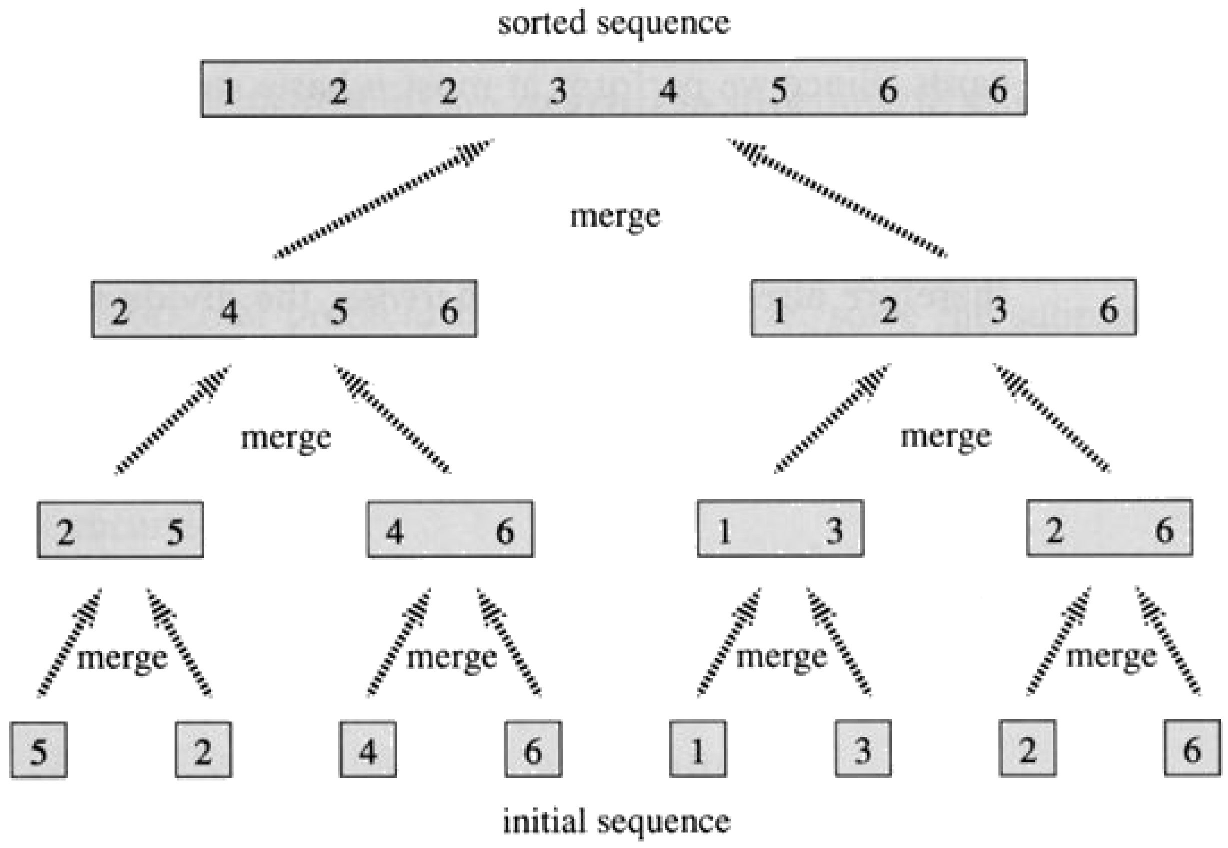
- Space efficiency: **in-place**. That is, $\Theta(1)$
- Stable sorting: **Yes**
- Best elementary sorting algorithm overall
  - Often used in Quicksort implementations
- Binary insertion sort?

Assignment 1: Submit on or before Feb 24th, Wednesday.

Q1. Compare the Quicksort lab exercise with the following modified version. Use median-of-three method to choose the pivot. It's the median of the leftmost, rightmost and the middle element of the array. Don't sort the subarrays of size smaller than or equal to 16 elements. Once Quicksort algorithm ends, apply the Insertion Sort algorithm for the whole nearly-sorted-array. Compare the execution time of the improved version with the exercise problem implemented in the lab for the same large input.

In the Blue Book, write the source code of the improved version and show the comparison of execution times.

Q2. Compare the Mergesort lab exercise with the bottom-up version demonstrated in the following image.



In the Blue Book, write the source code of the bottom-up version and show the comparison of execution times.

Channa Bankapur @ PES UNIVERSITY

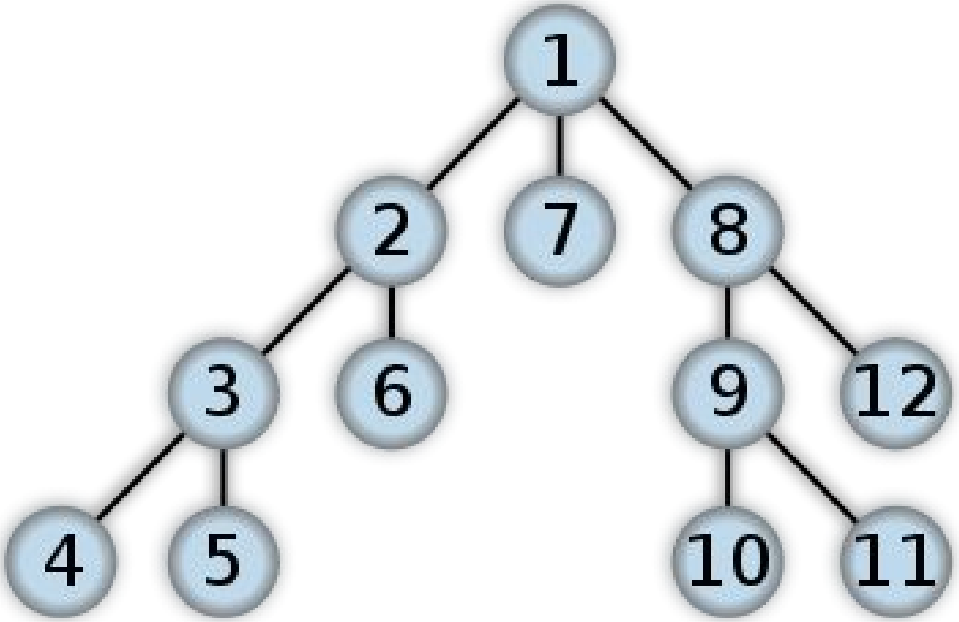
Graph Traversal Algorithms:

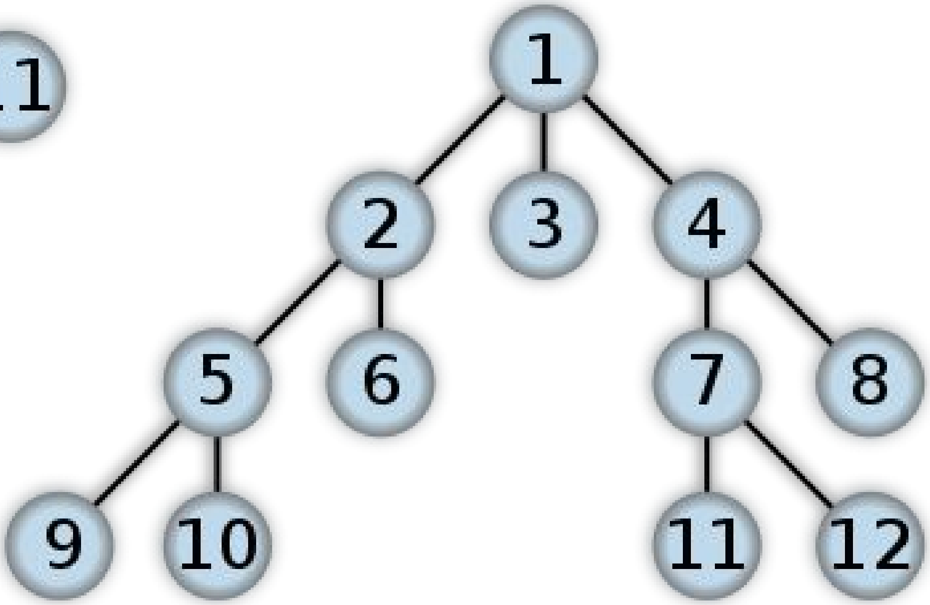
Many problems require processing all graph vertices (or edges) in a systematic way.

Based on the order of visiting the vertices:

- Depth-first search (DFS)
  - Uses Stack behavior

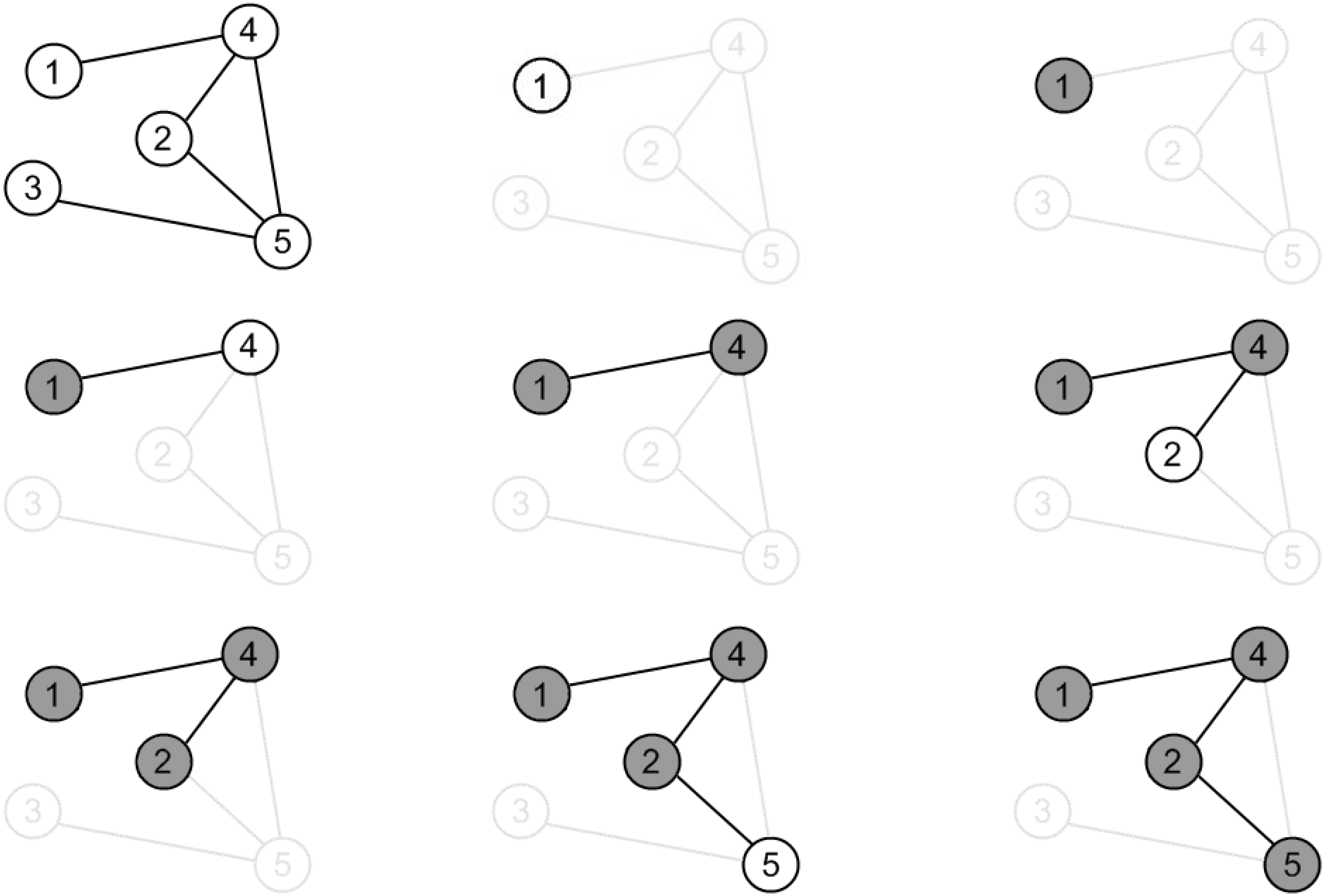
- Breadth-first search (BFS)
  - Uses Queue behavior

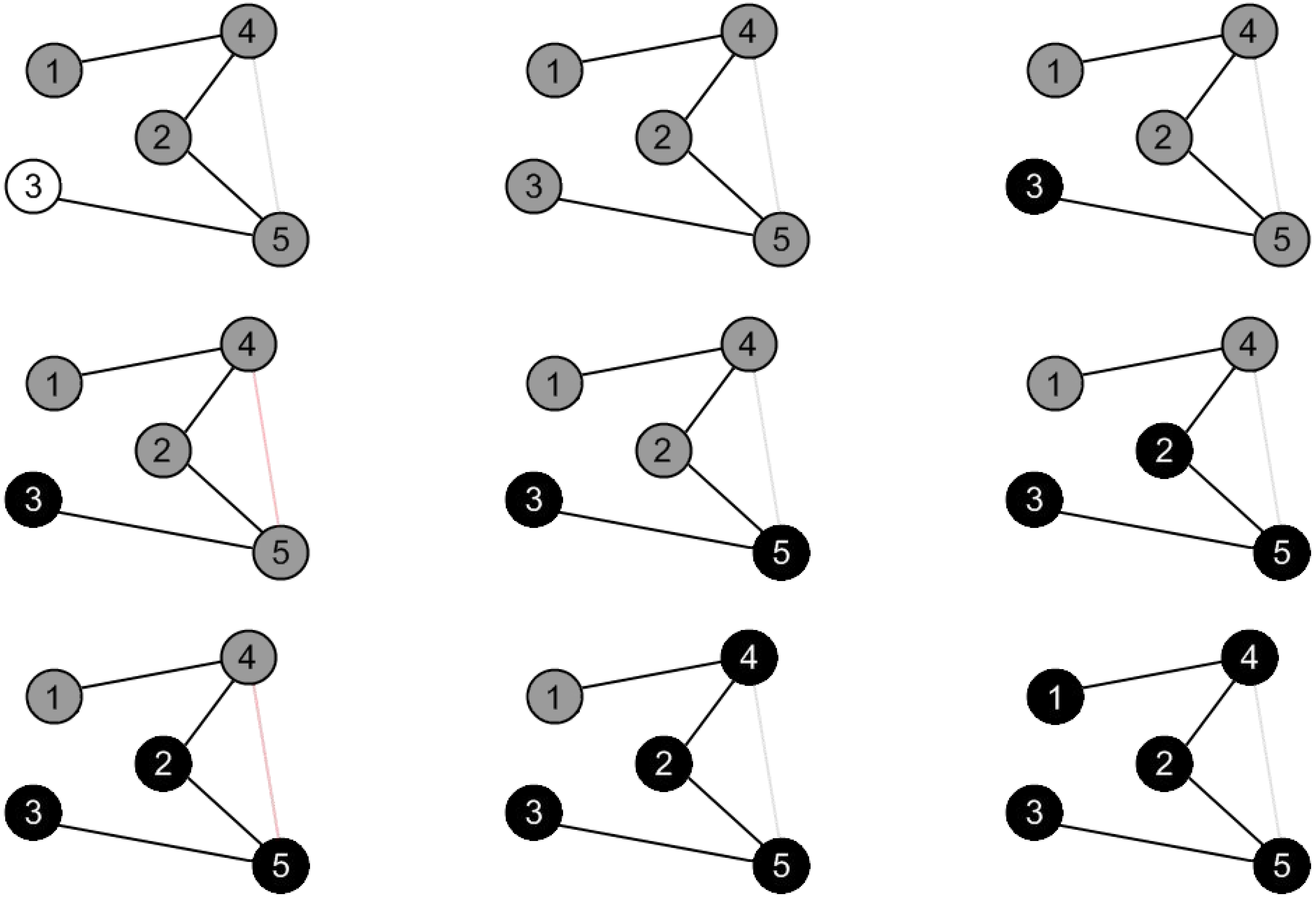

1
2
7
8
3
6
9
12
4
5
10
11

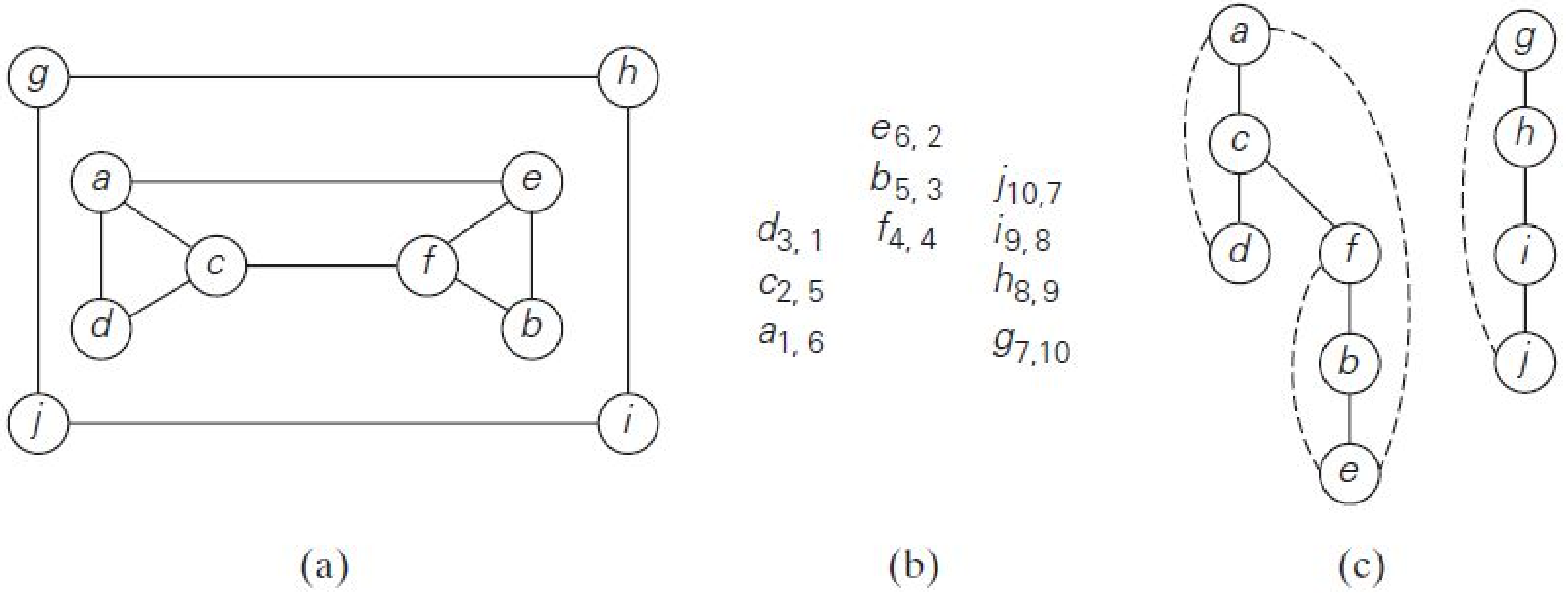

1
2
3
4
5
6
7
8
9
10
11
12

```
Algorithm DFS(G)
   Mark each vertex in v with 0
   count ← 0
   for each vertex v in V
      if (v is marked with 0)
         dfs(v)

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

```
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
            $dfs(w)$
```







(a) Graph

(b) Stack of the DFS Traversal

(c) DFS Forest

i. Tree edges

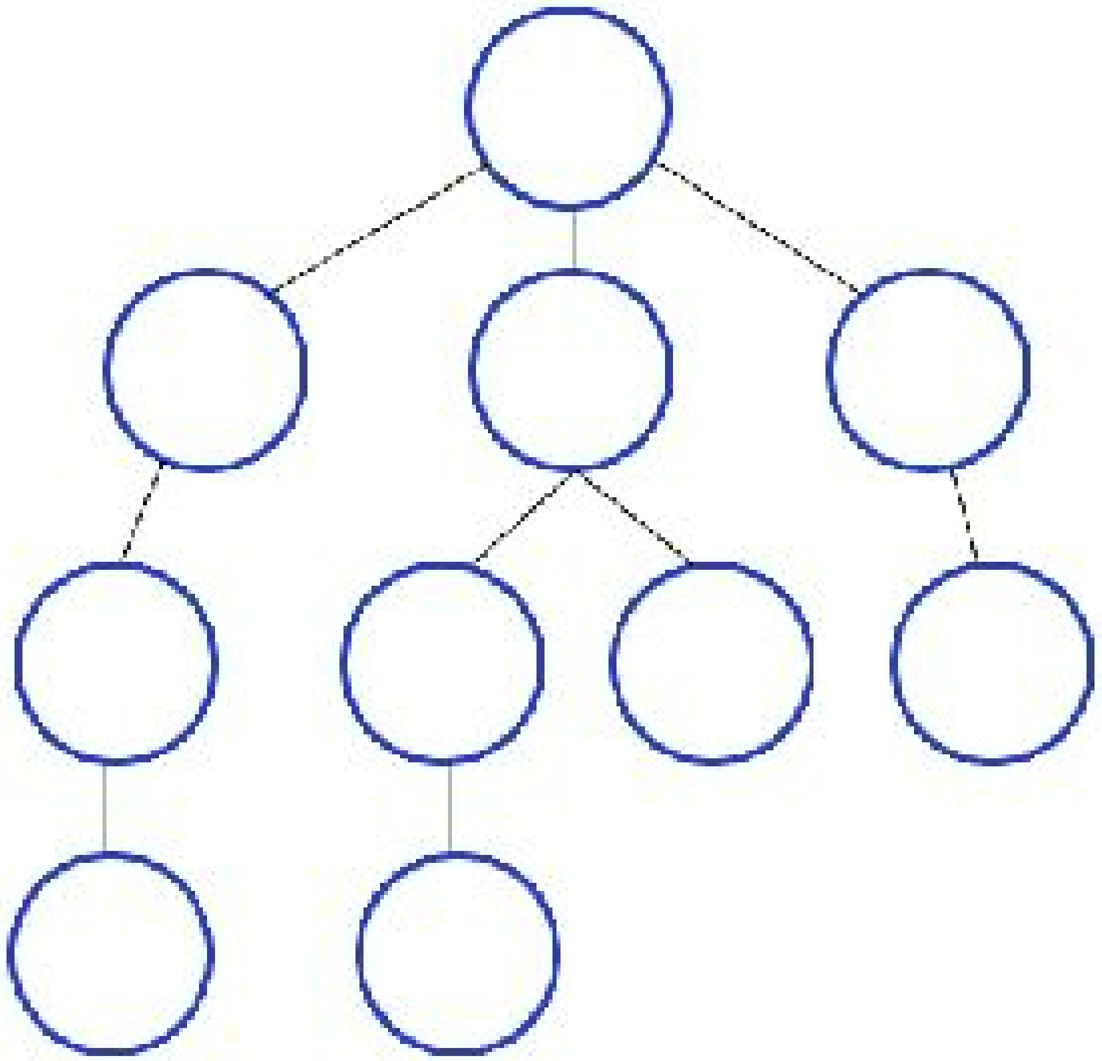
ii. Back edges

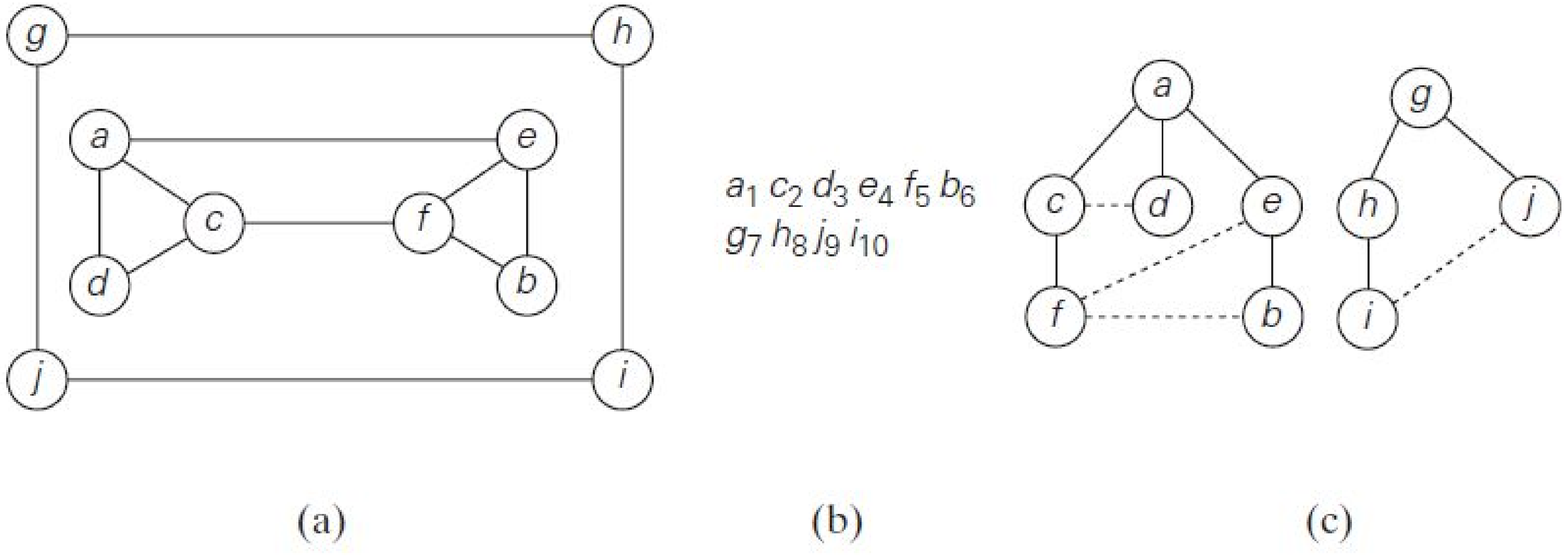
```
Algorithm BFS(G)
   Mark each vertex in v with 0
   count ← 0
   for each vertex v in V
      if(v is marked with 0)
         bfs(v)

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

Algorithm BFS(G)

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

```
ALGORITHM  $BFS(G)$
    //Implements a breadth-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 are 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 numbers them 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
```





(a) Graph

(b) Queue of the BFS Traversal

(c) BFS Forest

i. Tree edges

ii. Cross edges

Time Efficiency of DFS(G):

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

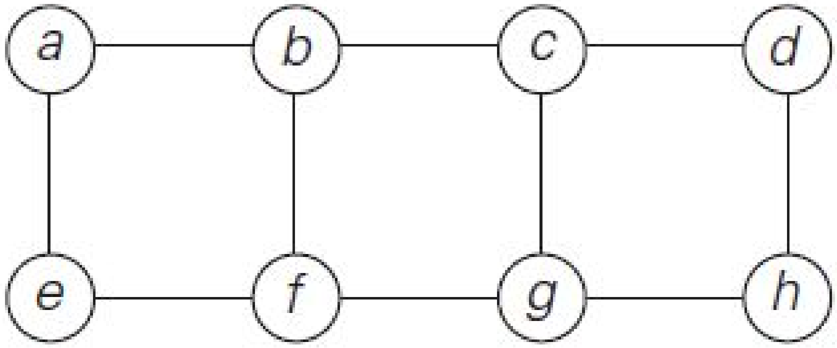
Adjacency Lists: $\Theta(|V| + |E|)$

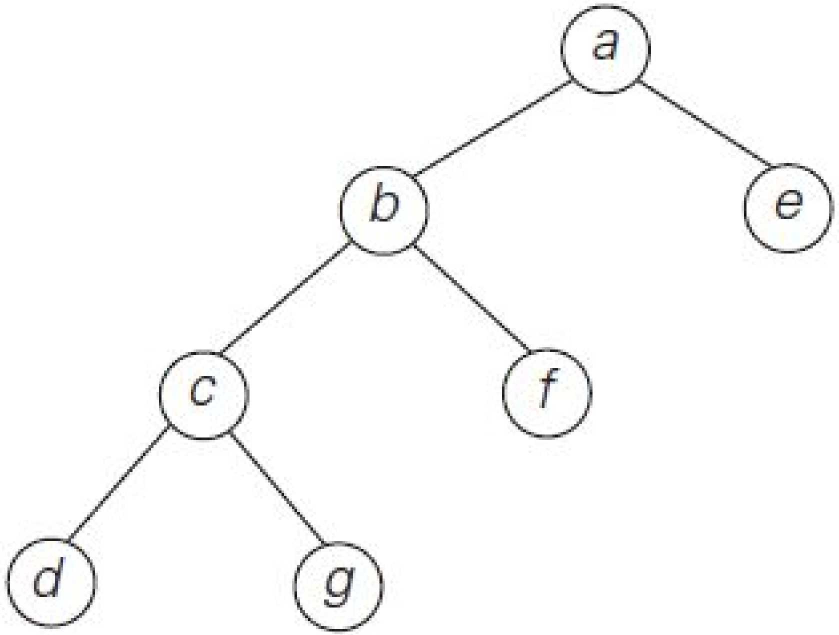
Time Efficiency of BFS(G):

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

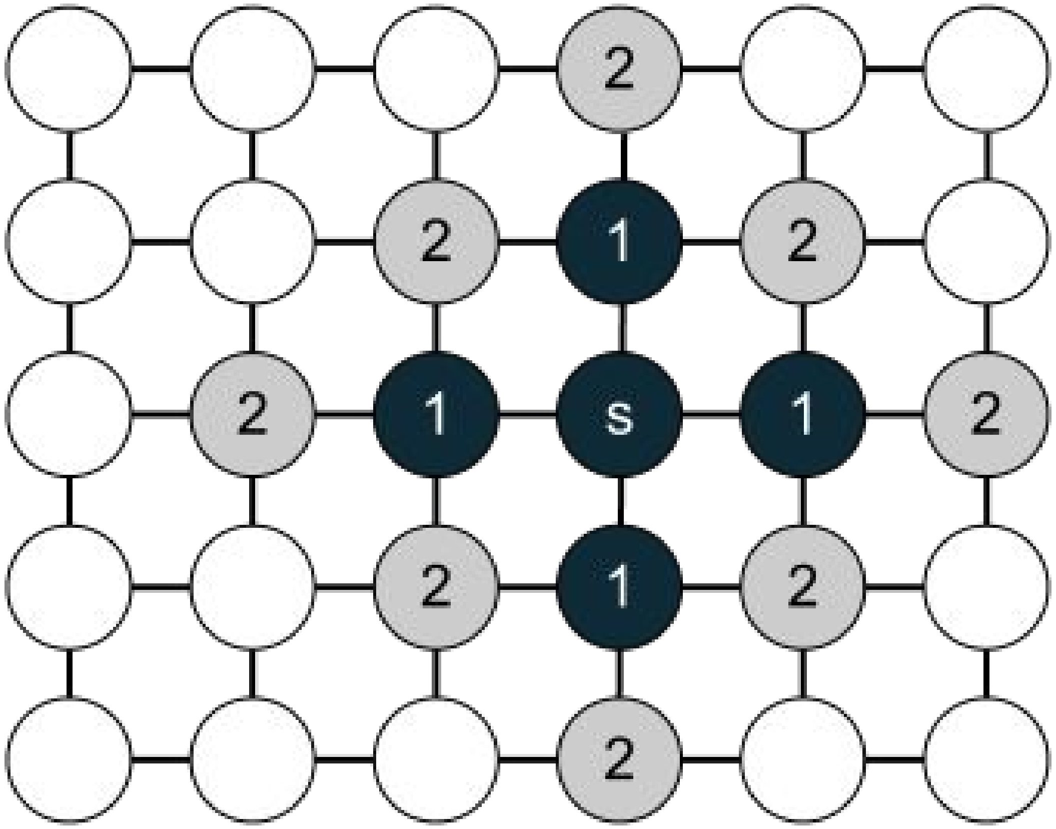
Adjacency Lists: $\Theta(|V| + |E|)$

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





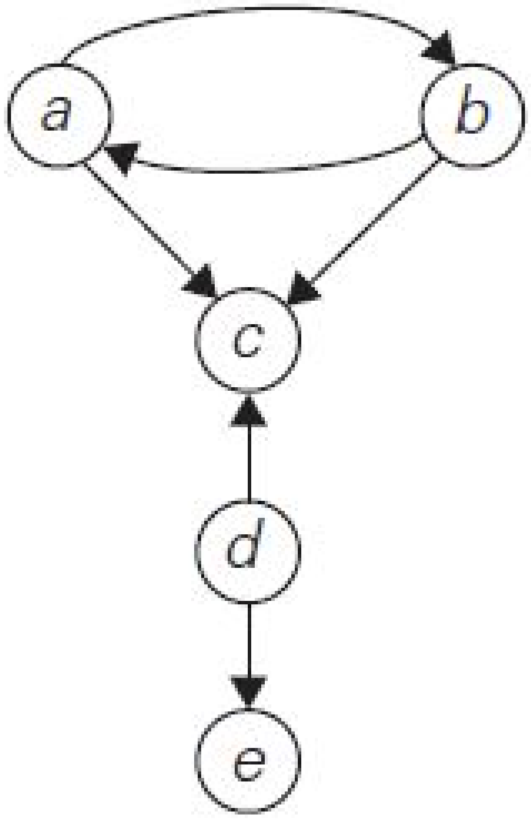
BFS: WHITE, GRAY, BLACK



Paths

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

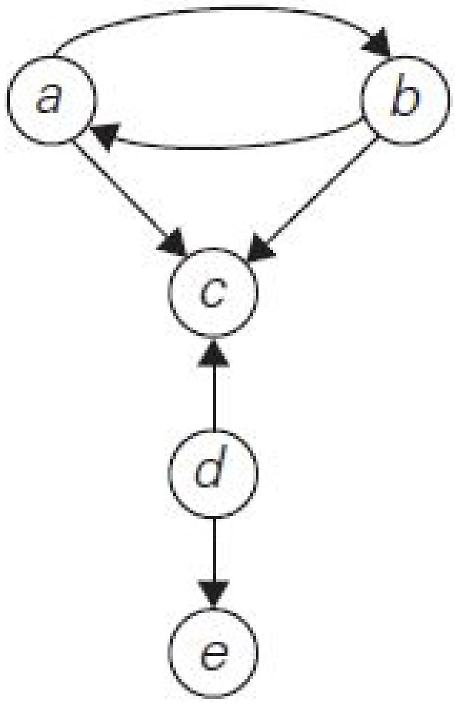
Draw a DFS forest for the given directed graph.



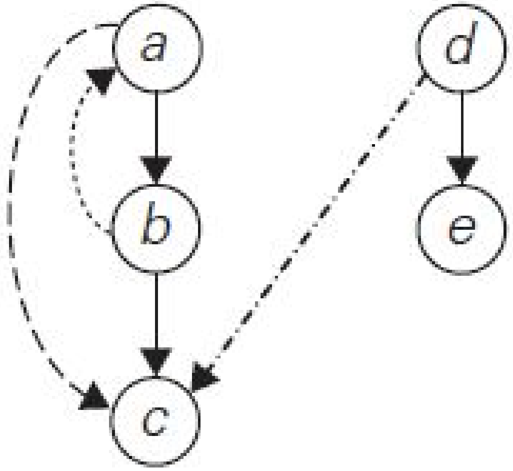
(a) DiGraph

(b) DFS Forest

i. Tree edges
ii. Back edges
iii. Forward edges
iv. Cross edges



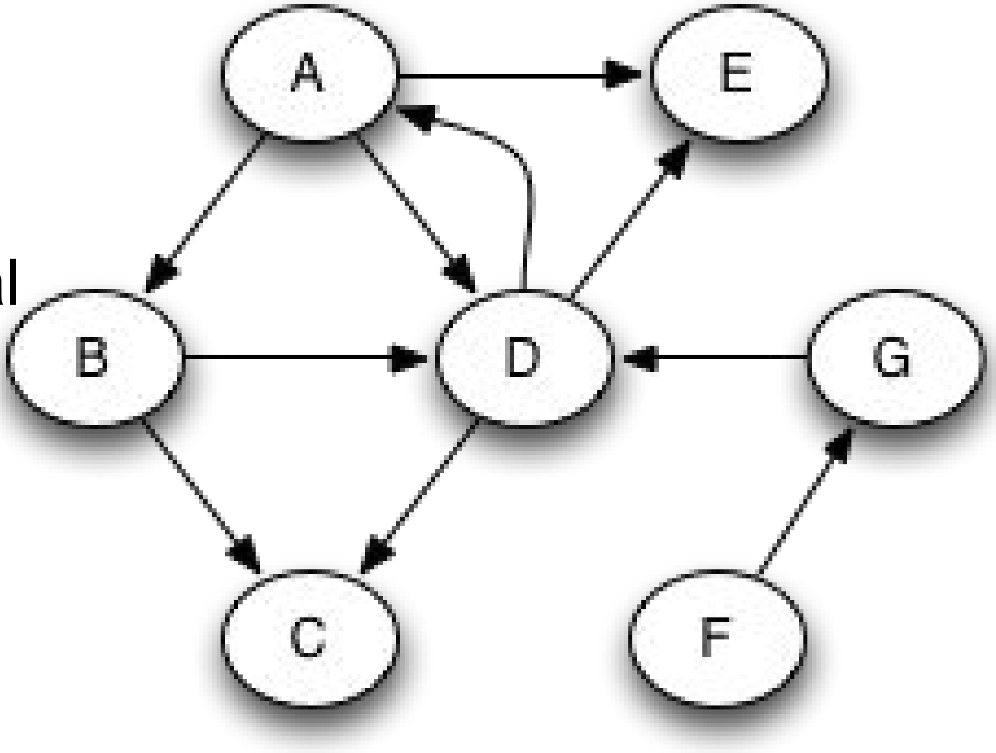
(a)



(b)

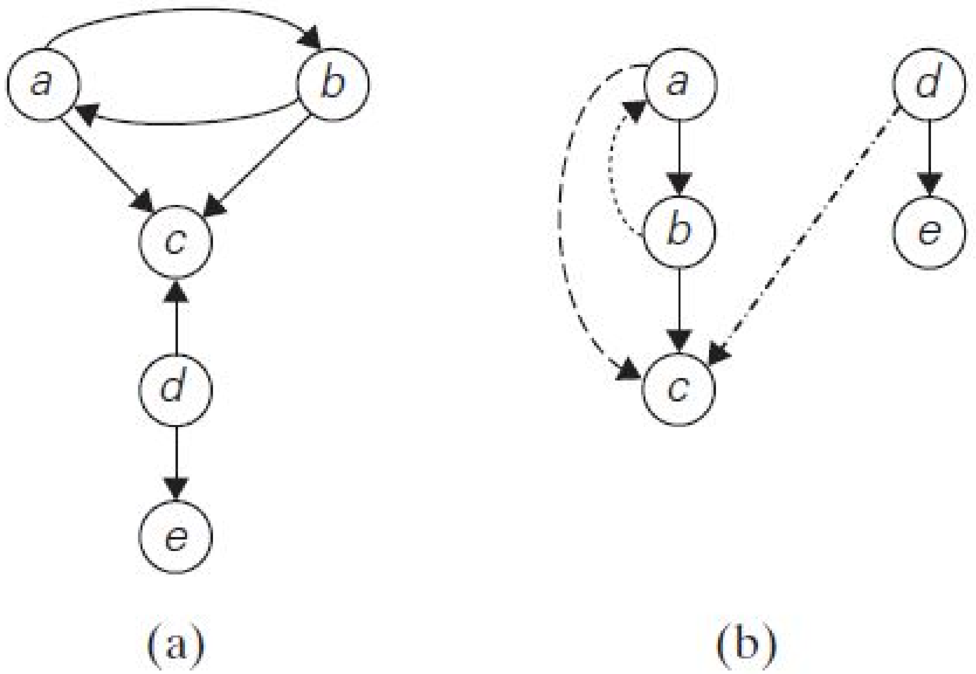
For the given graph, write

- Stack of the DFS Traversal
- DFS Forest
- Queue of the BFS Traversal
- BFS Forest


$$\begin{tikzcd}
A \arrow[rr] \arrow[d] \arrow[rd] & & E \\
B \arrow[r] \arrow[rd] & D \arrow[lu, bend right] \arrow[ru] \arrow[d] & G \arrow[l] \\
& C & F \arrow[u]
\end{tikzcd}$$

Directed cycle: The presence of back edge indicates that the digraph has a directed cycle.

DAG (**directed acyclic graph**)**:** If a DFS forest of a digraph has no back edges, then the digraph is a **dag**.



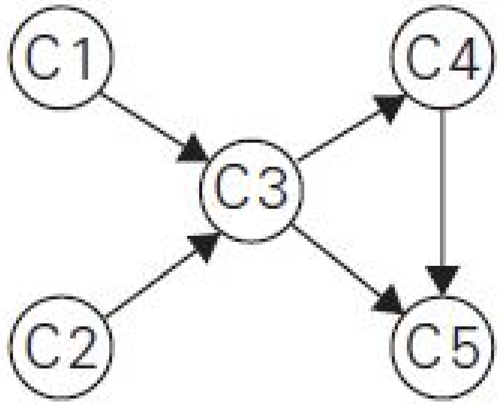
Topological Sorting:

(aka Toposort, Topological Ordering)

What do you know about **Topological Sorting** from a pre-requisite 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.

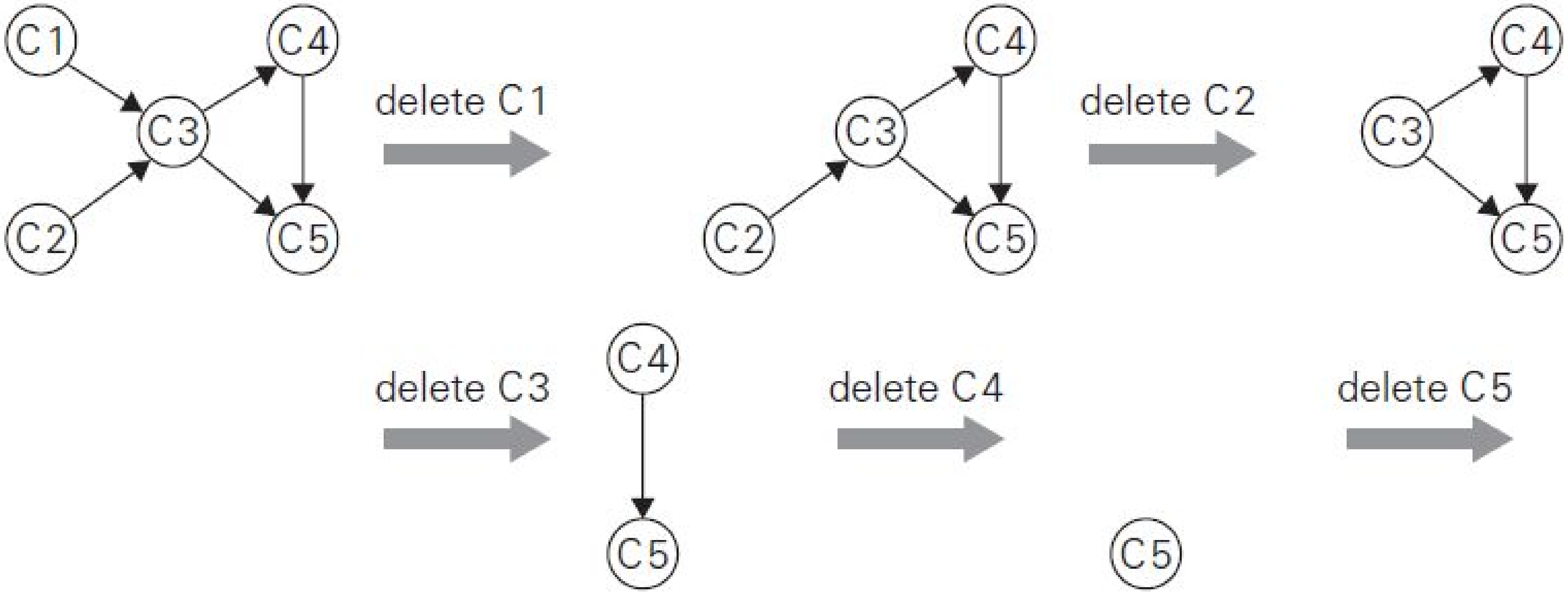
A digraph has a topological sorting iff it is a **dag**.



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

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

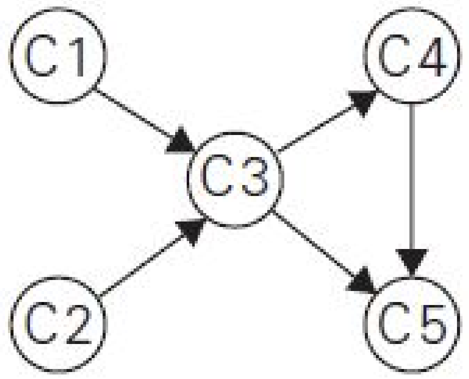
Source-removal algorithm for finding **Topological Sorting**



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)
```

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



(a)

$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

(b)

The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:

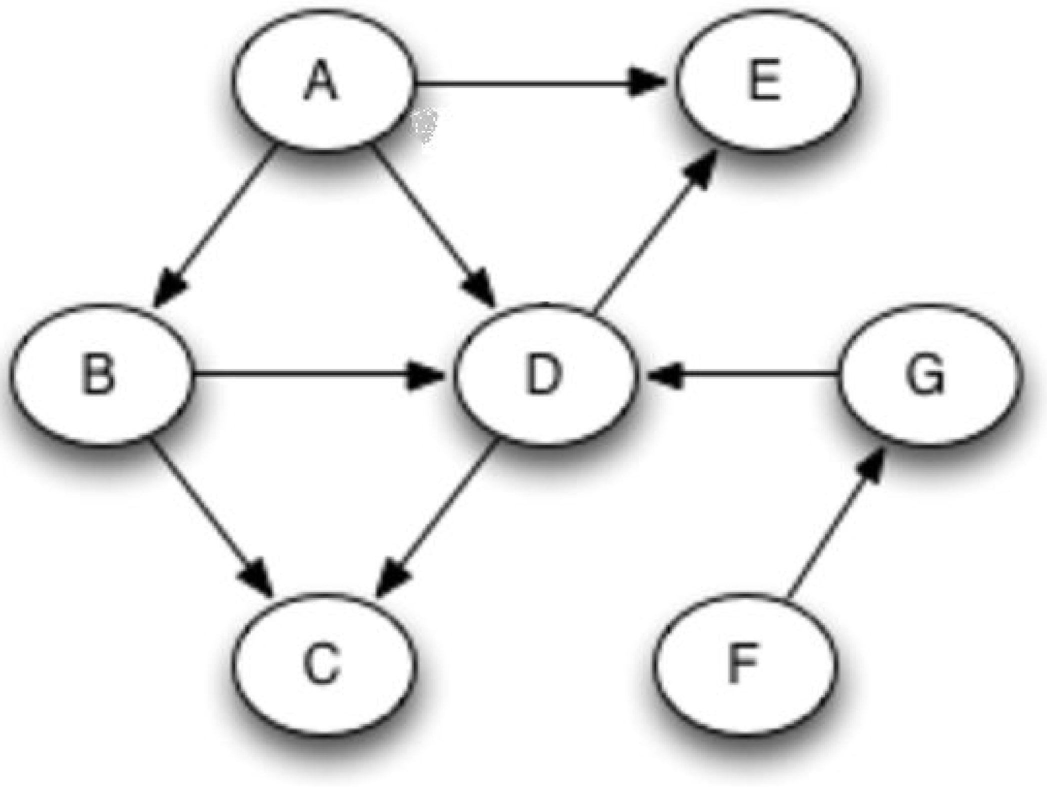
$$\begin{tikzcd} C2 \arrow[rr, bend right] & C1 \arrow[r] & C3 \arrow[r] \arrow[rr, bend right] & C4 \arrow[r] & C5 \end{tikzcd}$$

(c)

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

Procedure visit(vertex v)
    if pushed(v) != 0 then return error (not a DAG)
    if popped(v) = 0 then
        pushed(v) = 1
        for each vertex m with an edge from v to m do
            visit(m)
        popped(v) = 1, pushed(v) = 0
        add v at head of L
```

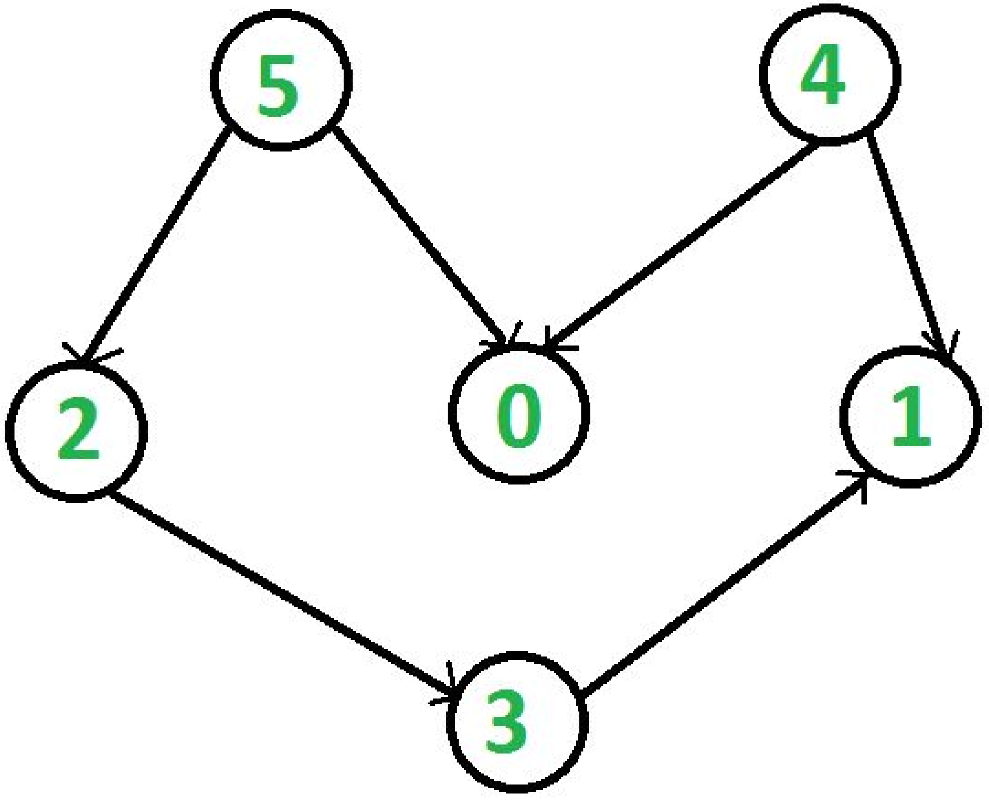
Topological Sorting of the vertices using

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



Topological Sorting of the vertices using

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

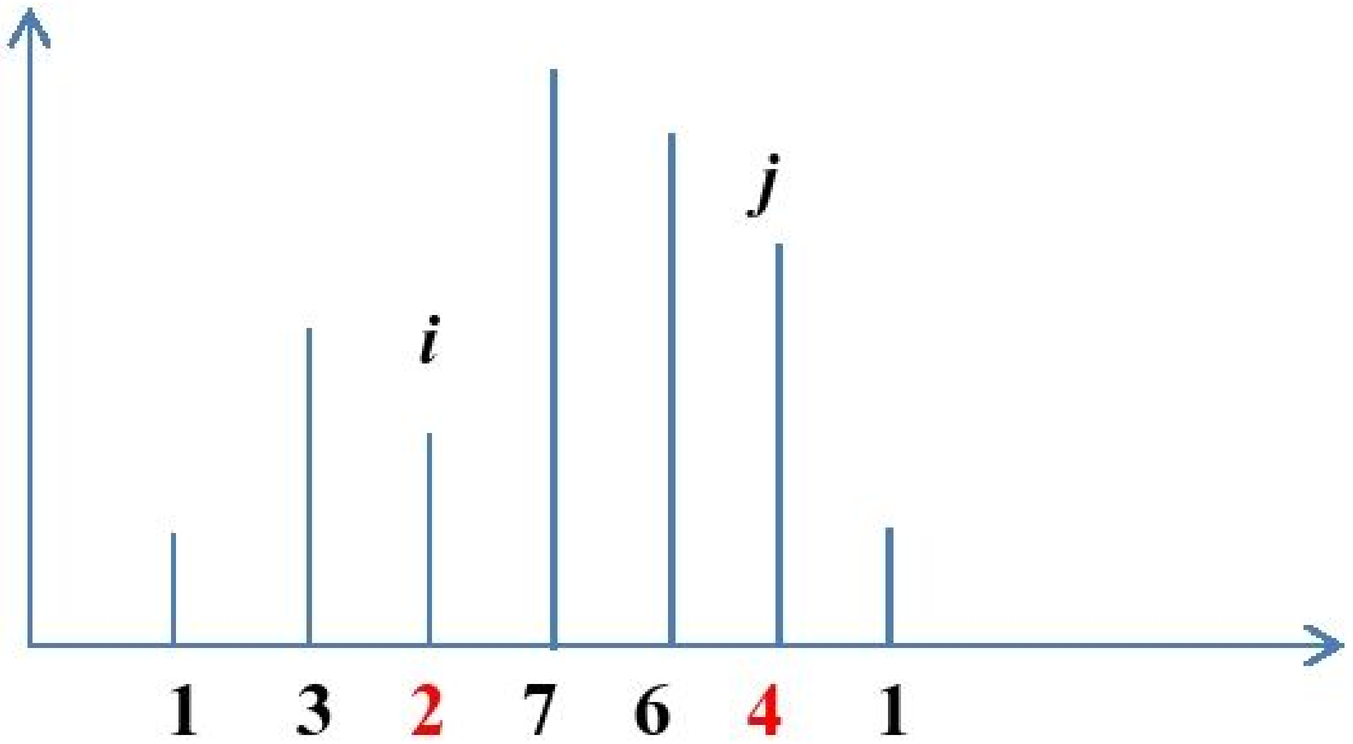


Generating Permutations:

123
132
213
231
312
321

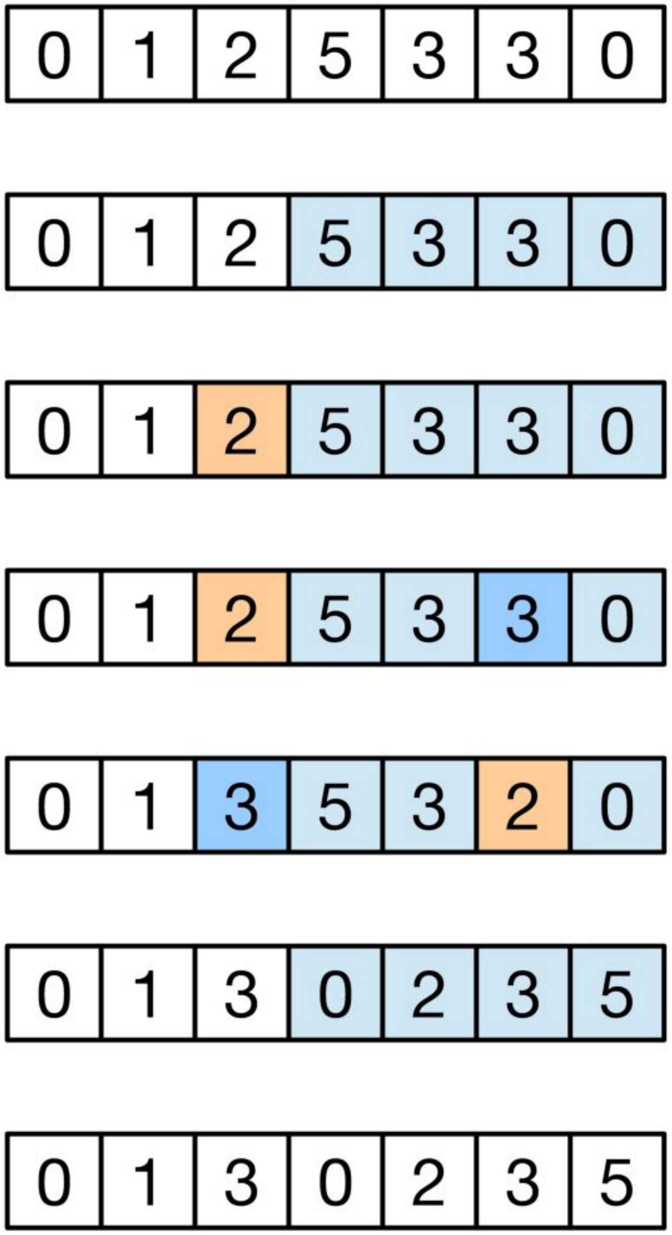
- Lexicographic order
  - the order in which they would be listed in a dictionary if the digits were interpreted as letters/characters.

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



```
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**

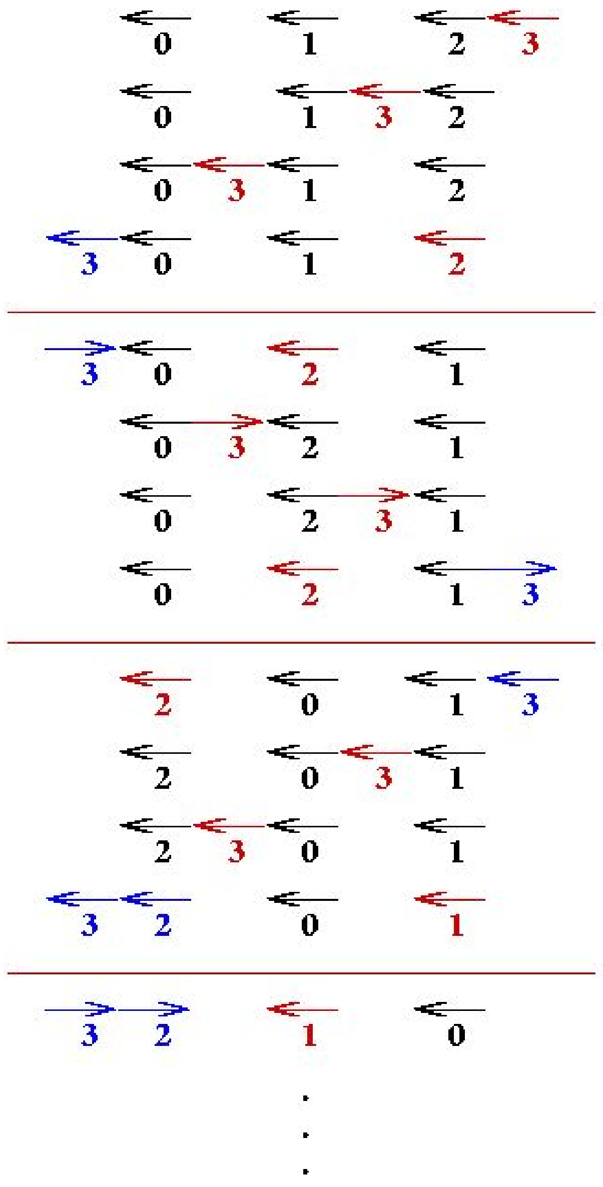
Solve it for input size of **(n-1)** and hence for **n**.

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



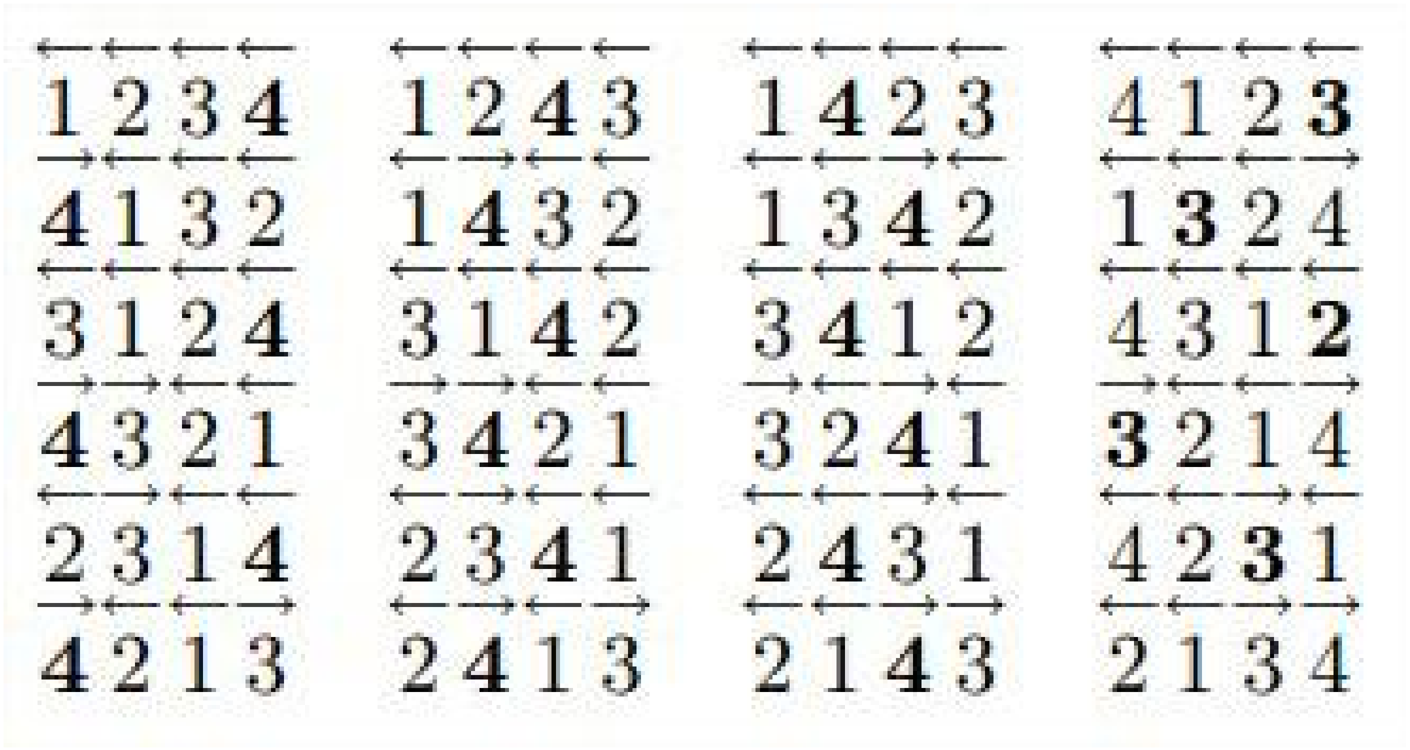
Move largest mobile element 3

Move largest mobile element 2
Reverse direction on all larger elements: 3

Move largest mobile element 2
Reverse direction on all larger elements: 3

Move largest mobile element 1
Reverse direction on all larger elements: 3, 2

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 **2^n** subsets.

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

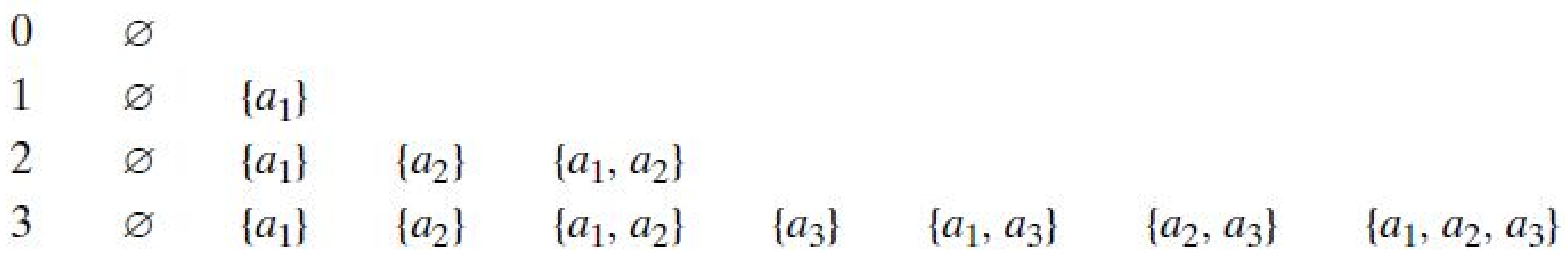
Any **decrease-by-one** idea?
# of subsets of { } = 2^0 = 1, which is **{ }** itself
Suppose, we know how to generate all subsets of {1,2,...,n-1}
Now, how can we generate all subsets of {1,2,...,n} ?

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:

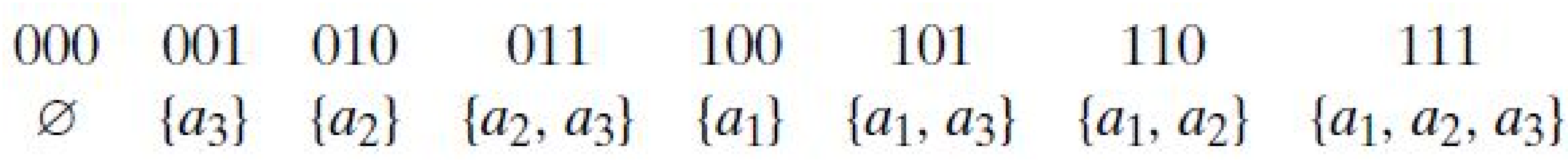
Knowing the binary nature of either having **n**th element or not, any idea involving binary numbers itself?

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 $\{a_1, a_2, …, a_n\}$.

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.

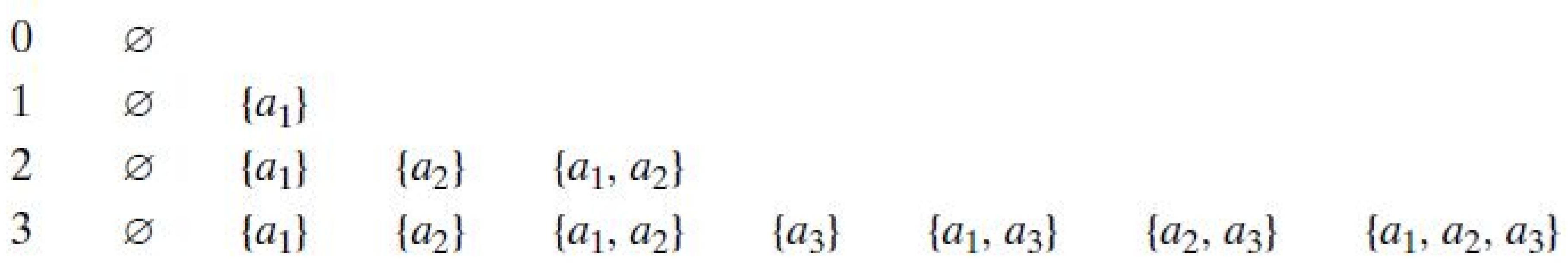


000	001	010	011	100	101	110	111
$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

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:

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\quad \overset{\overset{K}{\updownarrow}}{A[m]}\quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}$$

Decrease-by-a-Constant-Factor Algorithms:

```
ALGORITHM  $BinarySearch(A[0..n-1], K)$
    //Implements nonrecursive binary search
    //Input: An array $A[0..n-1]$ sorted in ascending order and
    //        a search key $K$
    //Output: An index of the array's element that is equal to $K$
    //        or $-1$ if there is no such element
    $l \leftarrow 0$;   $r \leftarrow n-1$
    while $l \le r$ do
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        if $K = A[m]$ return $m$
        else if $K < A[m]$ $r \leftarrow m-1$
        else $l \leftarrow m+1$
    return $-1$
```

Decrease-by-a-Constant-Factor Algorithms:

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	$\underline{2080}$
		3250

Decrease-by-a-Constant-Factor Algorithms:

Multiplication à la Russe:
(aka Russian peasant method)

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

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

$$1 \cdot m = m$$

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*** identically looking coins, one of which is fake. 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. Assume that the fake coin is known to be lighter than the genuine ones.

Decrease-by-a-Constant-Factor Algorithms:

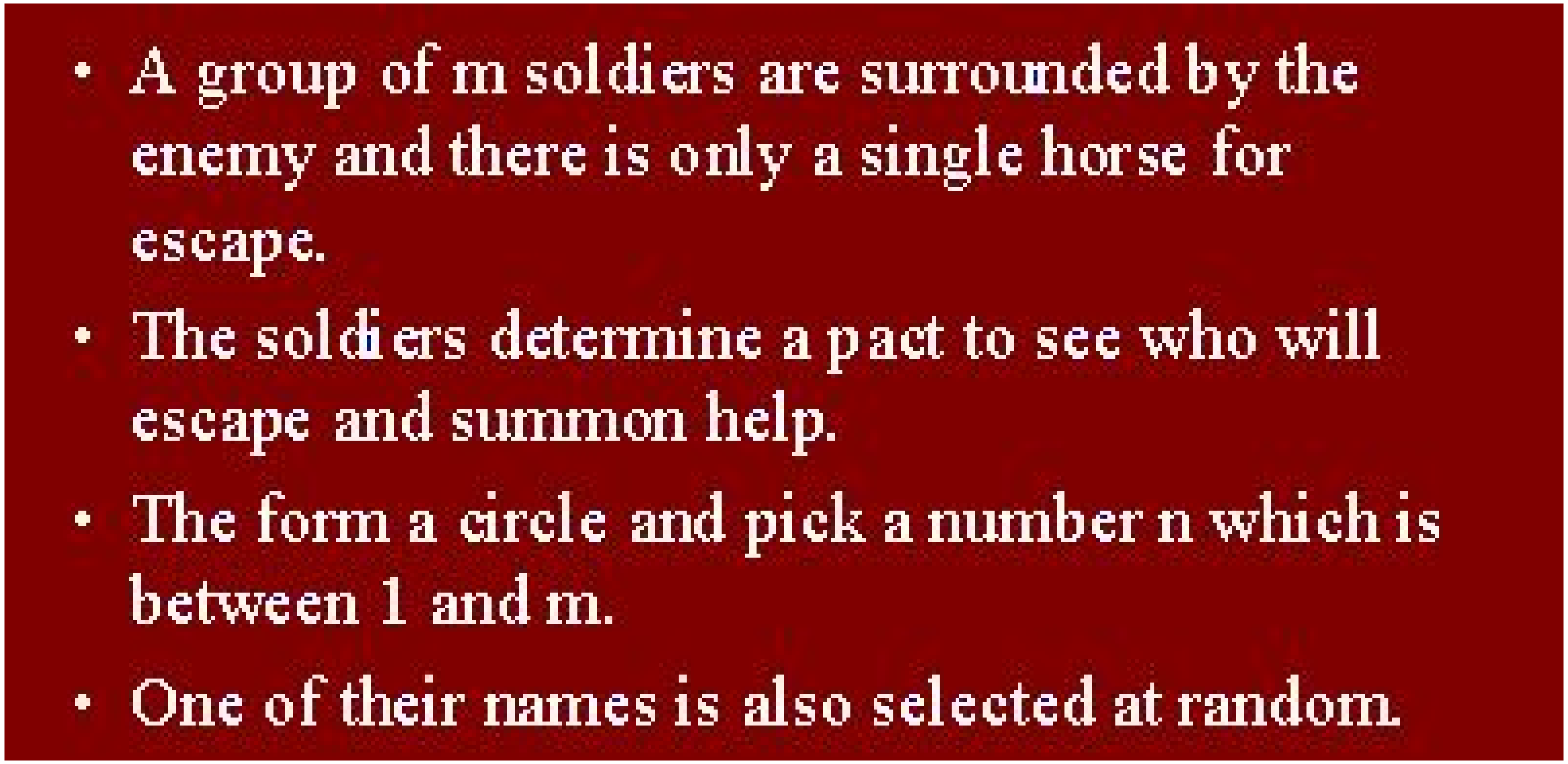
Fake-Coin Problem:

1. Decrease-by-a-factor of 2 algorithm

2. Decrease-by-a-factor of 3 algorithm

Josephus Problem:

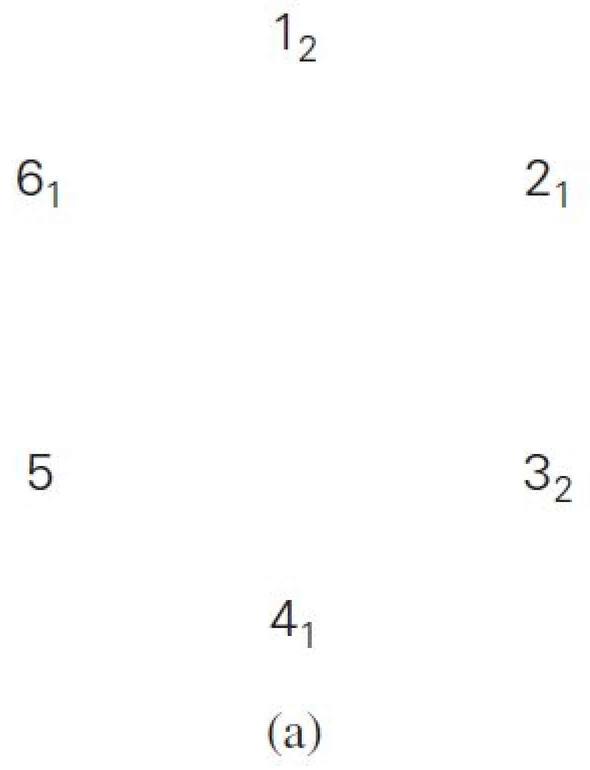
Let n people be numbered from 1 to n stand in a circle. Starting the count from 1, we eliminate every second person until only one survivor is left. The problem is to determine the survivor's number J(n).



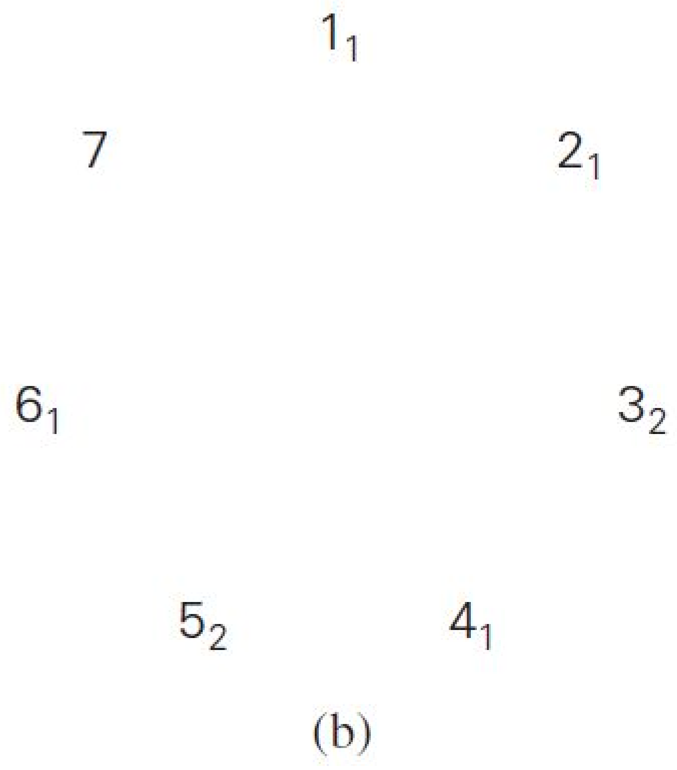
- A group of m soldiers are surrounded by the enemy and there is only a single horse for escape.
- The soldiers determine a pact to see who will escape and summon help.
- The form a circle and pick a number n which is between 1 and m.
- One of their names is also selected at random.

Josephus Problem:

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



(a)



(b)

Josephus Problem:

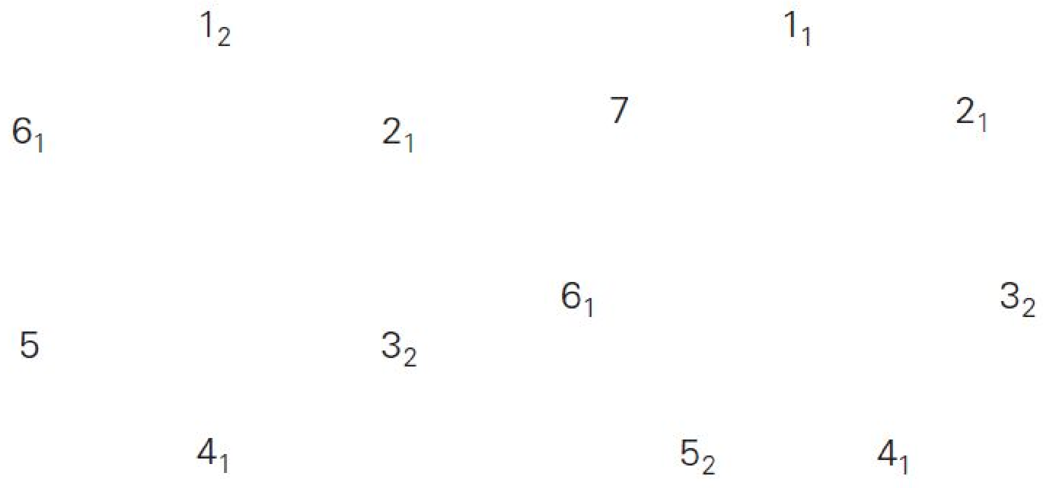
1	1	**5**	**5**	1	3	**7**	**7**	1	1	1	**9**
2	3	9		2	5	11		2	3	5	
3	**5**			3	**7**			3	5	**9**	
4	7			4	9			4	7		
5	9			5	11			5	**9**		
6				6				6	11		
7				**7**				7			
8				8				8			
9				9				**9**			
10				10				10			
				11				11			
								12			

Josephus Problem:

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

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

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



(a) (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 a step everyday toward reaching your goal!

</ End of Chapter 5 - Decrease-n-Conquer >