

Course – Analysis and Design of Algorithms

UNIT 2 – Part 2



Unit 2: Decrease-and-Conquer

- Insertion Sort
- Topological Sorting
- Algorithms for Generating Combinatorial Objects
- Decrease-by-a-Constant-Factor Algorithms: Binary Search
- Variable-Size-Decrease Algorithms: Computing Median and the Selection Problem

Unit 2: Decrease-and-Conquer

- Introduction

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 its smaller instance.**

- Once such a relationship is established, it can be exploited either top down or bottom up.
- Top down approach leads naturally to a recursive implementation.
- The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the *incremental approach*.

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 its smaller instance.**

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

- ☐ Decrease by a constant
- ☐ Decrease by a constant factor
- ☐ Variable size decrease

Decrease-and-Conquer

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

Decrease by a constant (or Decrease by **One**)

- The Size of an instance is reduced by same constant (usually by one) at each iteration of the algorithm.

Decrease by a constant factor (or Decrease by **half**)

- The size of a problem instance is reduced by same constant factor (usually by half) on each iteration of the algorithm.

Variable size decrease

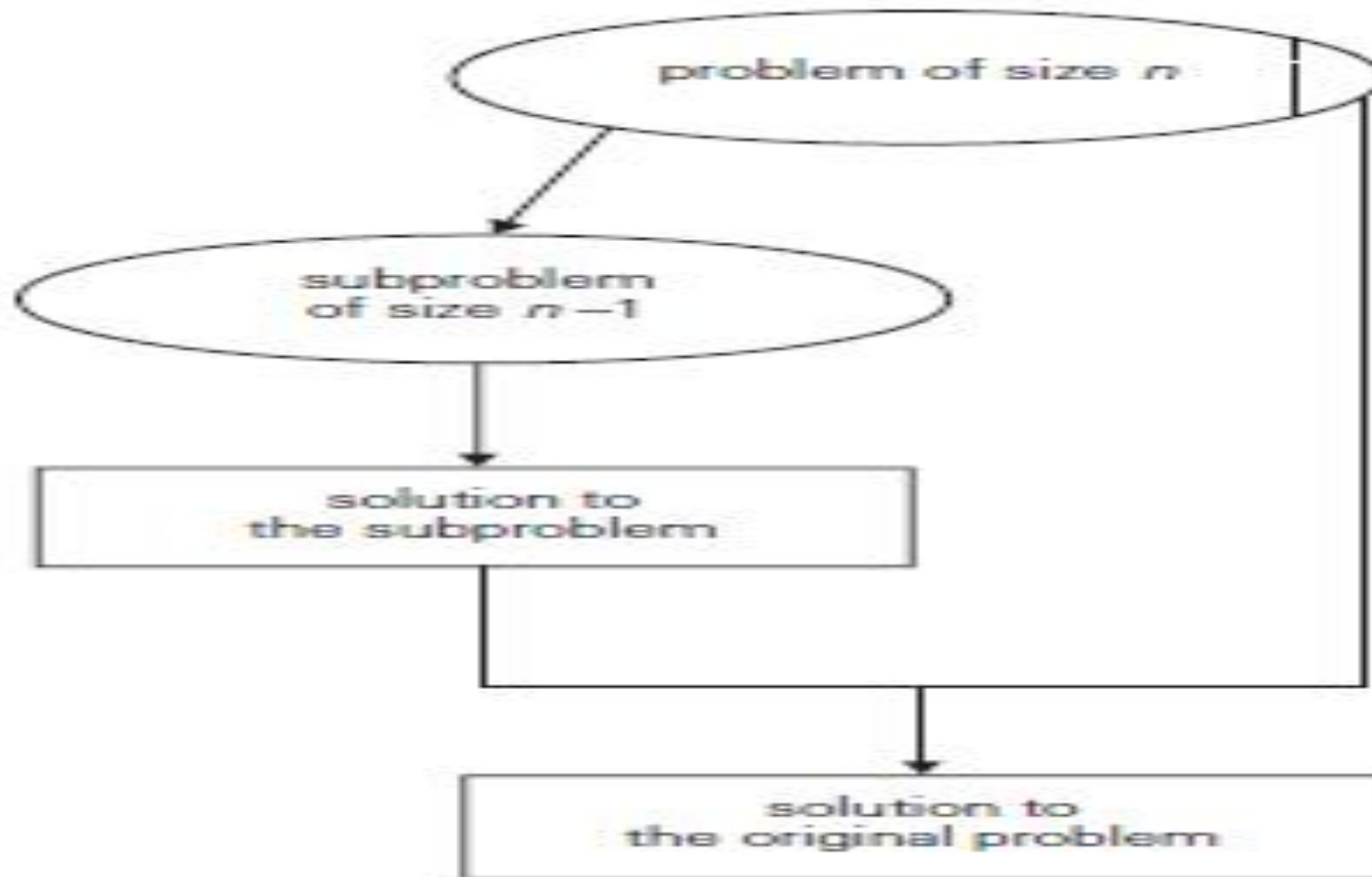
- A size reduction pattern varies from one iteration to another

Three major variations of decrease-and-conquer

- **Decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one.
- Example: we can **compute a^n recursively** (top-down approach) using **$a^n = a^{n-1} * a$** . The recurrence relation to compute a^n using decrease-and-conquer method is

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

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

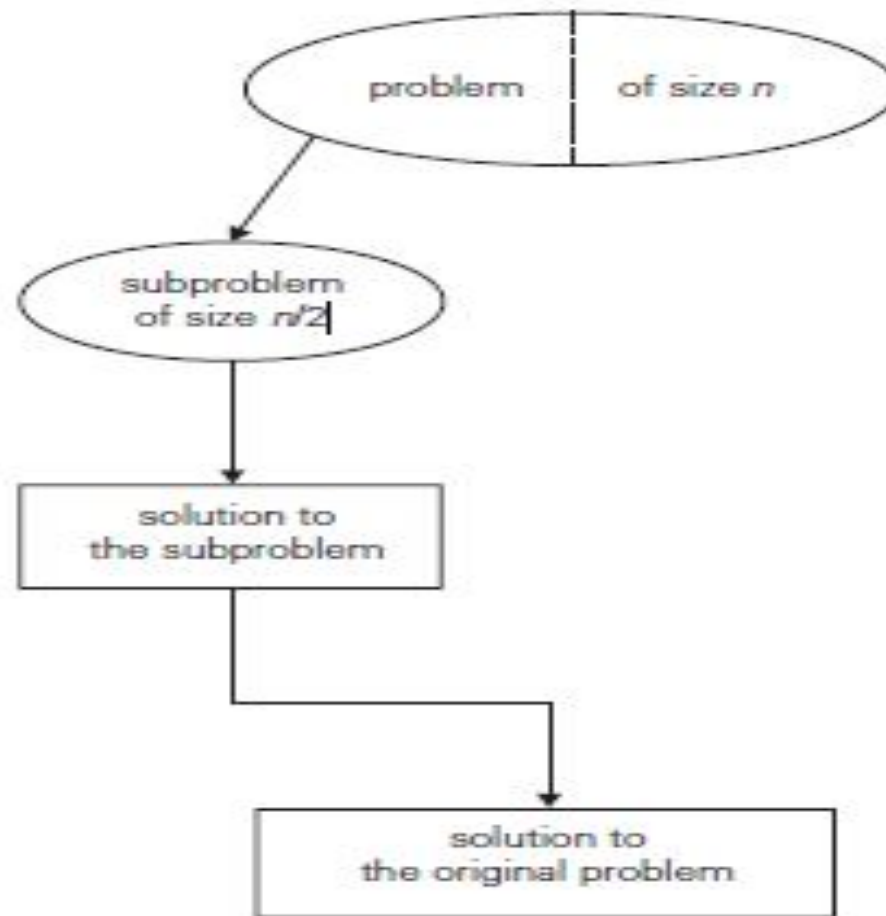


Three major variations of decrease-and-conquer

- **Decrease-by-a-constant-factor** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.
- Example: To compute a^n , the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

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



Three major variations of decrease-and-conquer

- **Variable-size-decrease** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation.

$$\text{gcd}(m,n) = \begin{cases} m & \text{if } n=0 \\ \text{gcd}(n, m \bmod n) & \text{Otherwise} \end{cases}$$

Computing gcd(6,10)

M	N	Description
6	10	Since n is not 0, $m \leftarrow n$ and $n \leftarrow m \% n$. So, i.e., $M \leftarrow 10$ and $N \leftarrow 6 \% 10$
10	6	Since n is not 0, $m \leftarrow n$ and $n \leftarrow m \% n$. So, $M \leftarrow 6$ and $N \leftarrow 10 \% 6 = 4$
6	4	Since n is not 0, $m \leftarrow n$ and $n \leftarrow m \% n$. So, $M \leftarrow 4$ and $N \leftarrow 6 \% 4 = 2$
4	2	Since n is not 0, $m \leftarrow n$ and $n \leftarrow m \% n$. So, $M \leftarrow 2$ and $N \leftarrow 4 \% 2 = 0$
2	0	Since $N = 0$ the GCD will be the value of M which is 2.

Question

Ferrying soldiers A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

Question

Ferrying soldiers: A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

Solution

First, the two boys take the boat to the other side, after which one of them returns with the boat. Then a soldier takes the boat to the other side and stays there while the other boy returns the boat. These four trips reduce the problem's instance of size n (measured by the number of soldiers to be ferried) to the instance of size $n-1$. Thus, if this four-trip procedure repeated n times, the problem will be solved after the total of $4n$ trips.

Applying Decrease by One Strategy

Ferrying Soldiers

Apply decrease-by-1 process:

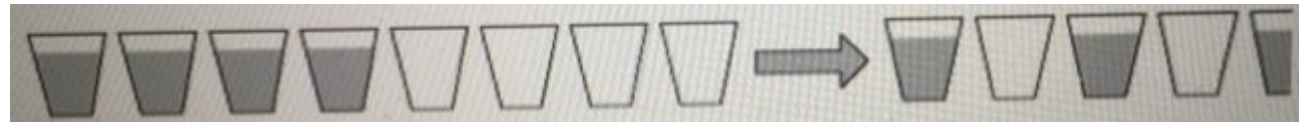
- Ferry *one soldier* to the far side, leaving boat and boys back at their initial positions
- If no soldiers remain, we have finished,
- Otherwise, ferry remaining $n-1$ soldiers

How many (one way) boat trips will it take to ferry one soldier?

A. 1 B. 2 C. 3 **D. 4** E. 5 F. 6

Homework Problem

- Alternating glasses There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves.



Assuming that the glasses are numbered left to right from 1 to $2n$, pour soda from glass 2 into glass $2n - 1$. This makes the first and last pair of glasses alternate in the required pattern and hence reduces the problem to the same problem with $2(n - 2)$ middle glasses. If n is even, the number of times this operation needs to be repeated is equal to $n/2$; if n is odd, it is equal to $(n - 1)/2$. The formula $\lfloor n/2 \rfloor$ provides a closed-form answer for both cases. Note that this can also be obtained by solving the recurrence $M(n) = M(n - 2) + 1$ for $n > 2$, $M(2) = 1$, $M(1) = 0$, where $M(n)$ is the number of moves made by the decrease-by-two algorithm described above. Since any algorithm for this problem must move at least one filled glass for each of the $\lfloor n/2 \rfloor$ nonoverlapping pairs of the filled glasses, $\lfloor n/2 \rfloor$ is the least number of moves needed to solve the problem.

Unit 2: Insertion Sort

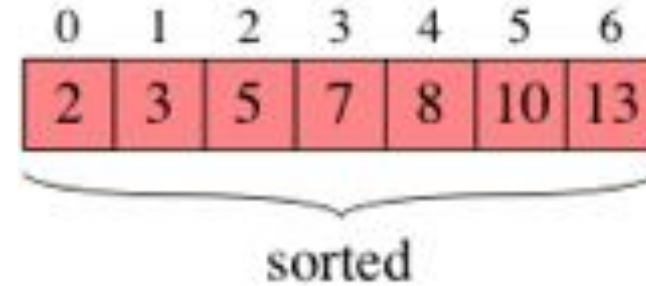
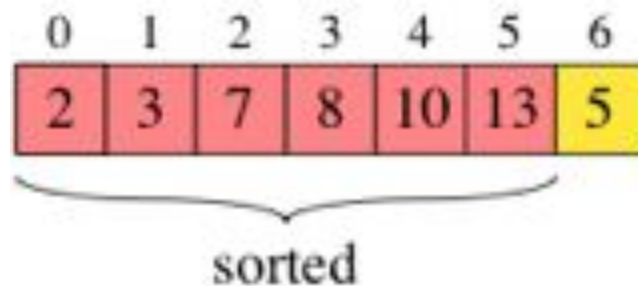
- Introduction

Insertion Sort: Introduction

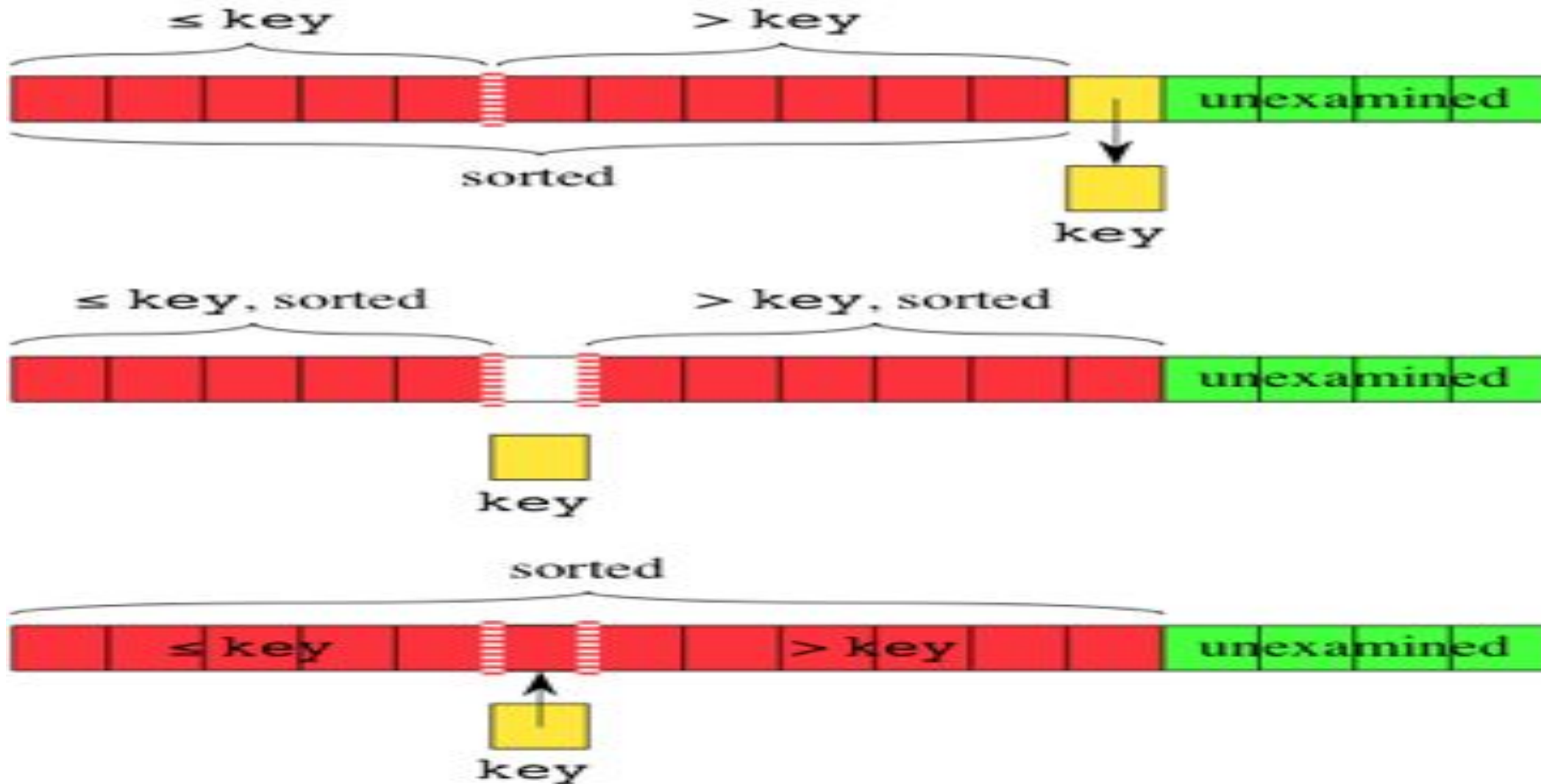
- This Sorting technique is very efficient if the elements to be sorted are partially arranged in ascending order.
- This technique is similar to the way a librarian keeps the books in the shelf.
- Initially all books will be placed in the shelf in increasing order of the access key number.
- When student returns the book to the librarian, he compares the access key of this book with other books in shelf one by one and inserts into the correct position so that all books are arranged in increasing order of their access key.

Insertion Sort: Logic

- Consider array of n elements to sort.
- We assume $a[i]$ is the item to be inserted.
 - Compare the item $a[i]$ from position $(i-1)$ down to 0. The item to be compared with $a[j]$, $(i-1) \geq j \geq 0$ where initially j is $(i-1)$.
 - As long as the item is less than $a[j]$, copy $a[j]$ to $a[j+1]$ and decrement j by 1.
 - As we decrement j by one, we should see that j should be greater than or equal to zero.
 - When the item is greater than $a[j]$ or j less than zero, insert item into $a[j+1]$
- If the above procedure is repeated for item $a[i]$ for $0 < i < n$, then all items in the array a will be sorted.



Insertion Sort: Logic

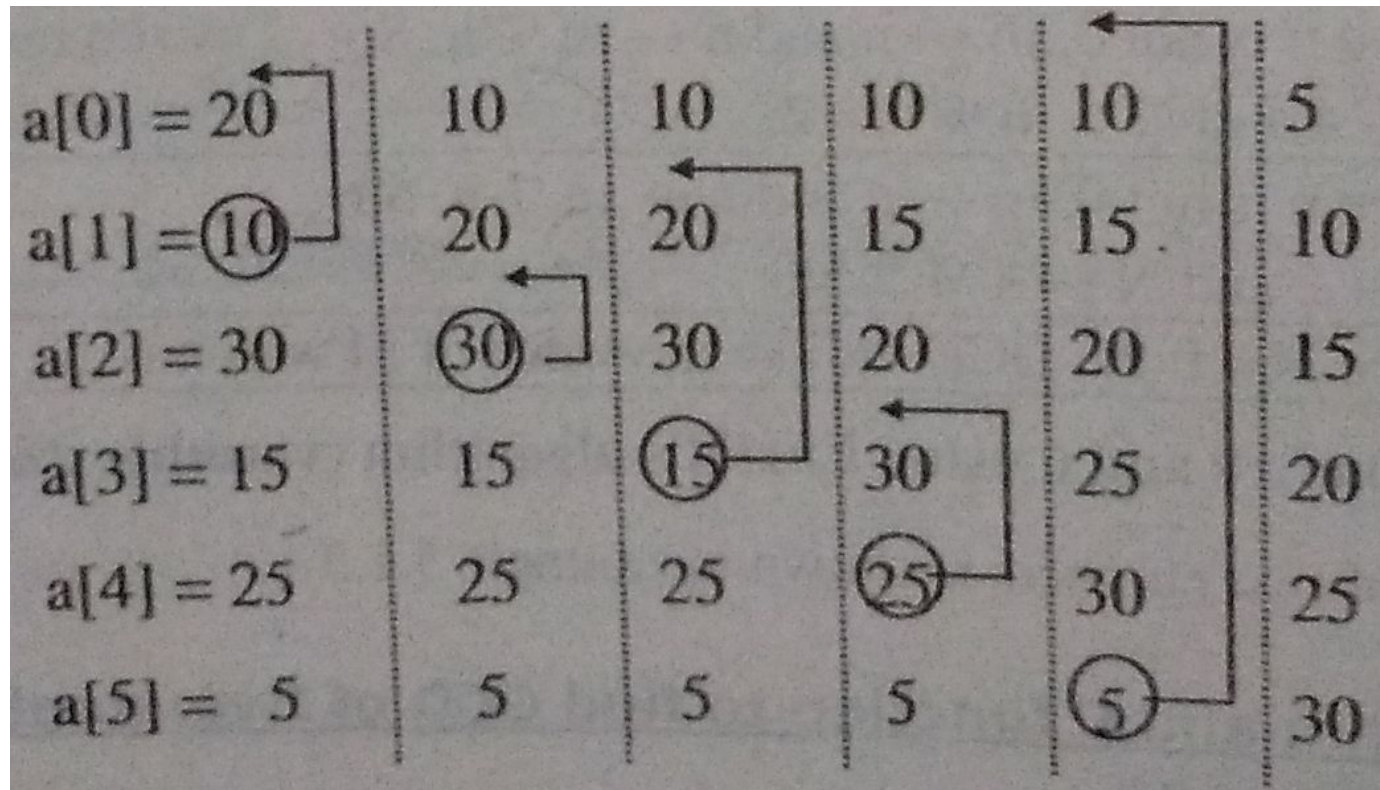


Insertion Sort: Example

- Sort the elements (20,10,30,15,25,5) using Insertion Sort

Insertion Sort: Example

- Sort the elements (20,10,30,15,25,5) using Insertion Sort



Question

- Sort the elements (89, 45, 68, 90, 29, 34, 17) using Insertion Sort

Question

- Sort the elements (89, 45, 68, 90, 29, 34, 17) using Insertion Sort

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

Homework Problem

- Apply insertion sort to sort the list E, X, A, M, P, L, E in alphabetical order.

<i>E</i>	<i>X</i>	<i>A</i>	<i>M</i>	<i>P</i>	<i>L</i>	<i>E</i>
<i>E</i>	X					
<i>E</i>	<i>X</i>	A				
<i>A</i>	<i>E</i>	<i>X</i>	M			
<i>A</i>	<i>E</i>	<i>M</i>	<i>X</i>	P		
<i>A</i>	<i>E</i>	<i>M</i>	<i>P</i>	<i>X</i>	L	
<i>A</i>	<i>E</i>	<i>L</i>	<i>M</i>	<i>P</i>	<i>X</i>	E
<i>A</i>	<i>E</i>	<i>E</i>	<i>L</i>	<i>M</i>	<i>P</i>	<i>X</i>

Insertion Sort: Algorithm

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** $(n - 1)$ {

$item \leftarrow A[i]$

$j \leftarrow (i - 1)$

while $j \geq 0$ **and** $A[j] > item$ {

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

$j \leftarrow j - 1$

 }

$A[j + 1] \leftarrow item$

}

Insertion Sort: Analysis

- The basic operation of the algorithm is the key comparison $A[j] > item$.
- The number of key comparisons in this algorithm obviously depends on the nature of the input.

Insertion Sort: Best case Analysis

- Best case occurs when the elements to be inserted is already sorted.

$$C(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = n - 1 = \Omega(n)$$

Insertion Sort: Worst case Analysis

- Worst case occurs when the condition $A[j] > item$ is executed maximum number of times. This situation occurs when the elements of the arrays are sorted in descending order.

$$\begin{aligned}C(n) &= \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} (i-1) - 0 + 1 = \sum_{i=1}^{n-1} i \\&= (n-1) + (n-2) + \dots + 3 + 2 + 1 \\&= \frac{(n-1)n}{2}\end{aligned}$$

So, if we consider $f(n) = \frac{(n-1)n}{2} = n^2/2 - n/2$, then

$$n^2/2 - n/2 \leq n^2 \quad \text{for } n \geq 0 \text{ which is of the form}$$

$f(n) \leq c_2 g(n)$ for $n \geq n_0$ where $c = 1$, $n_0 = 0$ $g(n) = n^2$. So, by definition, $f(n) = O(g(n)) = O(n^2)$.

So, time complexity of insertion sort in the worst case is $O(n^2)$

Insertion Sort: Average Case Analysis

Let us assume that two elements are already sorted and we have to insert the 3rd item at the appropriate position. There are 3 possible places where the item can be inserted. Let us take all possible cases one by one.

Case 1: $i = 2$ (position where the item has to be inserted) and $\text{Item} = a[2] = 13$

A	10	12	13
	0	1	2
		↑	↑
		j	i

It is clear from figure that the item is already in the correct position and has to be inserted in the same position. Since item 13 is greater than $a[j]$, control comes out of the while loop and the while loop is executed only once. So, the total number of times the while loop is executed is 1.

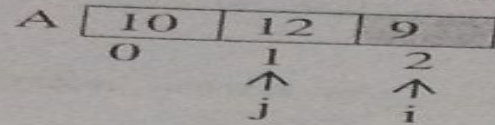
Case 2: $i = 2$ (position where the item has to be inserted) and $\text{Item} = a[2] = 11$

A	10	12	11
	0	1	2
		↑	↑
		j	i

Note that the item 11 has to be compared with 12 and 10 and should be inserted between 10 and 12 which results in the while loop to be executed 2 times. So, the total number of times the while loop is executed is 2.

Insertion Sort: Average Case Analysis

Case 3: $i = 2$ (position where the item has to be inserted) and Item = $a[2] = 9$



In this case, item 9 is compared with 12 and 10 and should be inserted before 10 which results in the while loop being executed 3 times.

Note that all these three cases have the same probability and the average number of times the while loop is executed is given by

$$\left(\frac{1 + 2 + 3}{3} \right) = 2$$

This result is true if we are inserting the 3rd element into the array. In general, to insert an item X with index i , in the correct position, the total number of times the while loop is executed is given by

$$\frac{\sum_{j=1}^i j}{i} = \frac{i(i+1)}{2 \cdot i} = \frac{i+1}{2} \qquad \frac{(1 + 2 + 3 + \dots + i)}{i}$$

It is clear from the algorithm that the index variable i starts from 1 to $n-1$. So, the total number of tests or comparisons made is given by

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \frac{i+1}{2} \\ \sum_{i=1}^{n-1} \frac{i+1}{2} &= \frac{1}{2} \sum_{i=1}^{n-1} i + 1 = \frac{1}{2} (2 + 3 + \dots + n) \\ &= \frac{1}{2} \left(\left[\frac{n(n+1)}{2} \right] - 1 \right) = \frac{1}{2} \left(\frac{n^2}{2} + \frac{n}{2} - 1 \right) \\ &= \frac{n^2}{4} + \frac{n}{4} - \frac{1}{2} \approx \frac{n^2}{4} \approx n^2 \end{aligned}$$

So, time complexity of insertion sort in the average case is $\Theta(n^2)$

Topological Sorting

Idea: For a directed Acyclic Graph, Topological sorting is a Linear ordering of vertices such that for every directed edge (u,v) vertex u comes before v in ordering.



Topological Sorting using the following two methods

- Source Removal Method
- Depth First search method

Topological Sorting

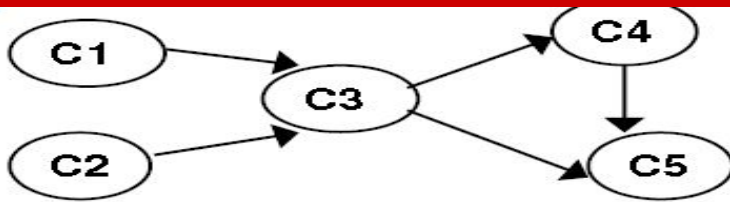
Source Removal Method:

Is based on direct implementation of decrease and conquer technique.

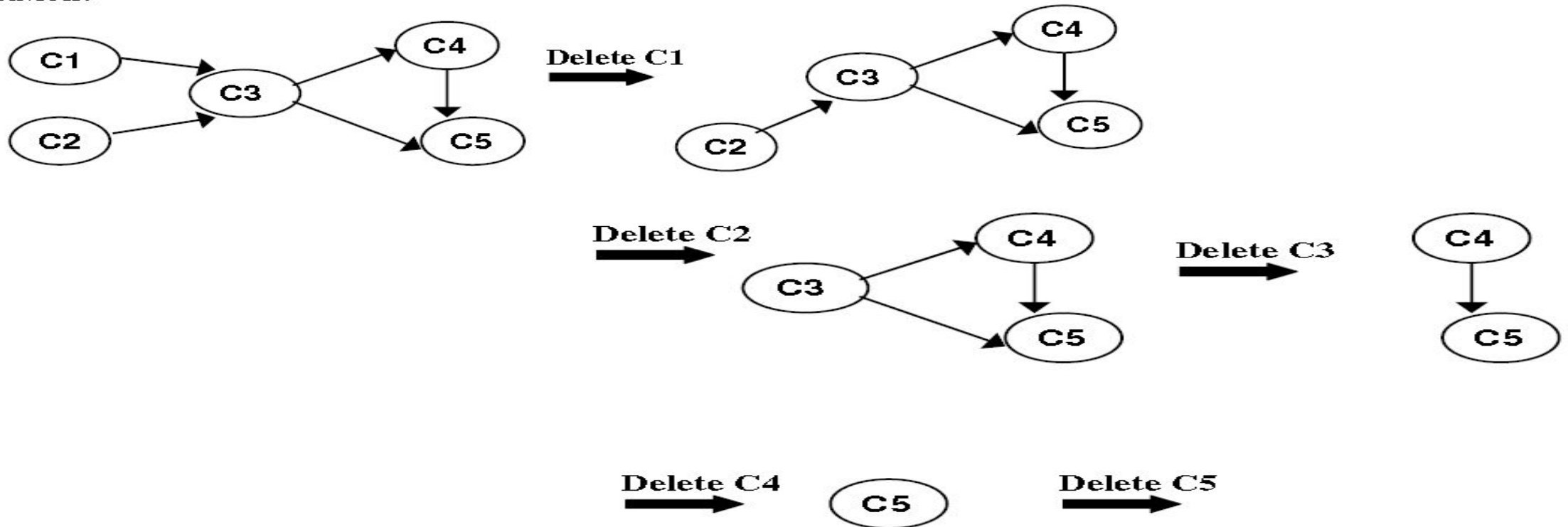
Repeatedly, identify in a remaining digraph a source, which is a **vertex with no incoming edge and delete it along with all edges outgoing from it.**

The order in which the vertices are deleted yields a solution to topological sorting problem.

Topological Ordering Using Source Removal method



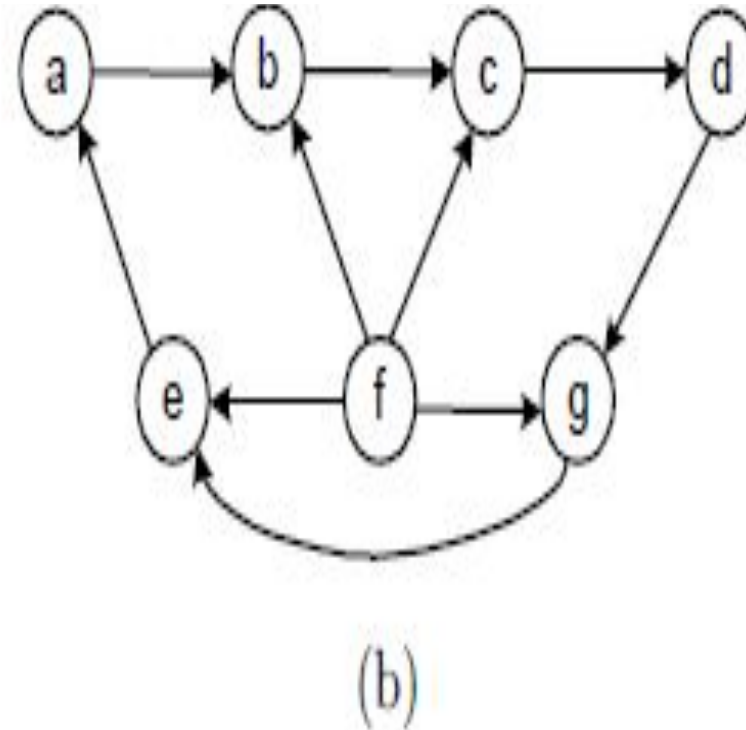
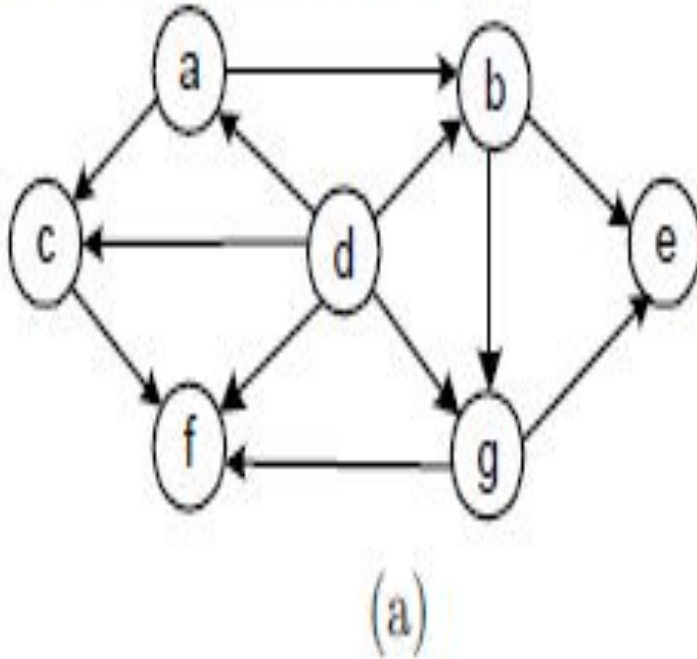
Solution:



The topological order is C1, C2, C3, C4, C5

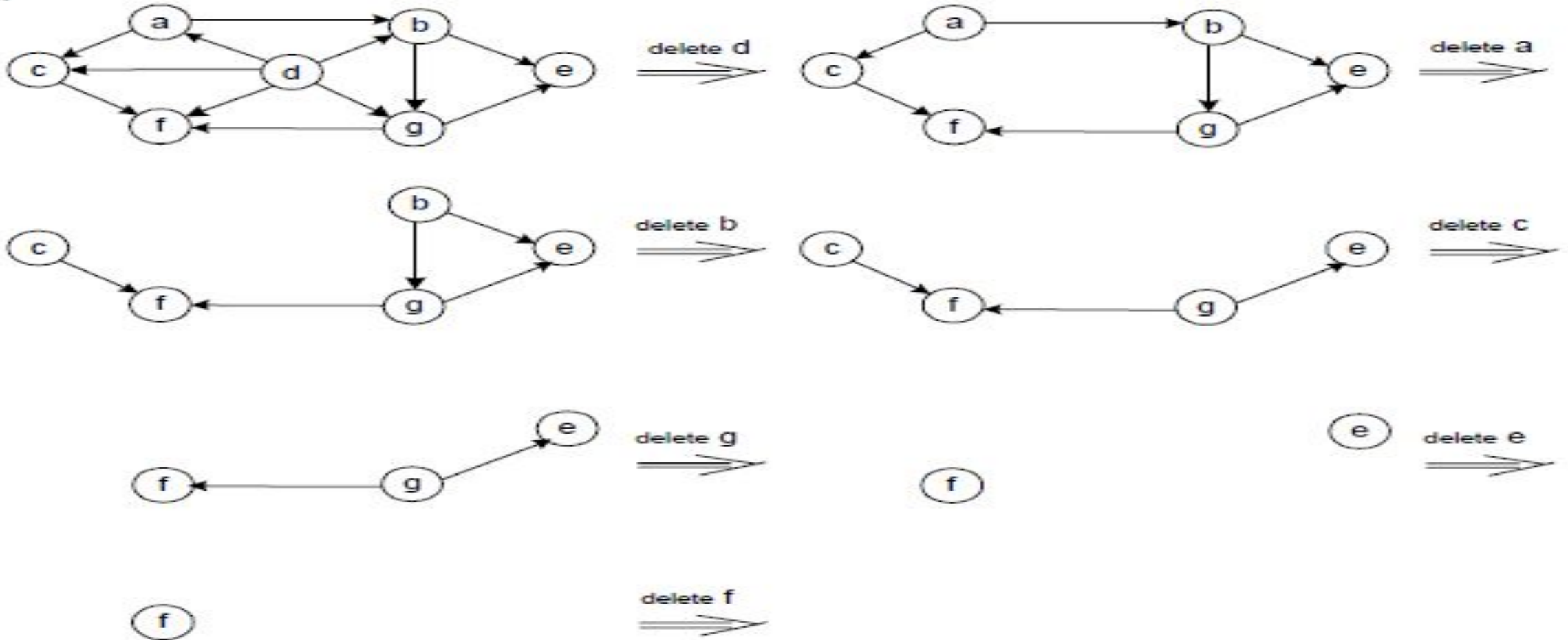
Question

- Apply the source-removal algorithm to the digraphs



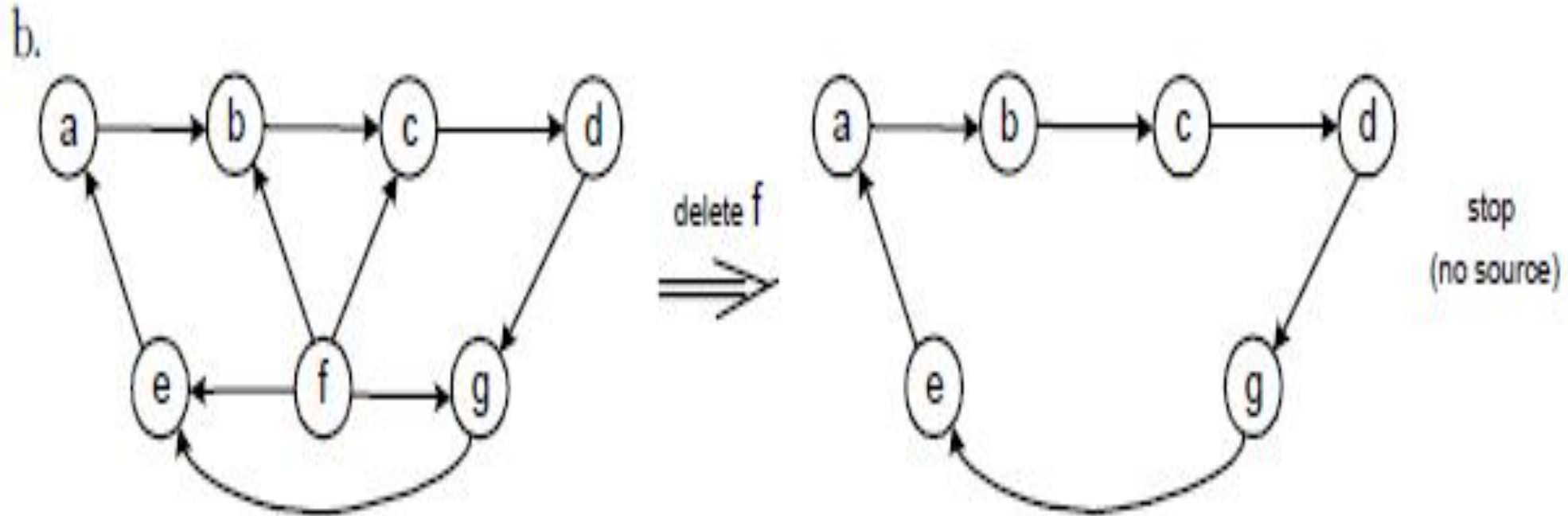
Answer

a.



The topological ordering obtained is *d a b c g e f*.

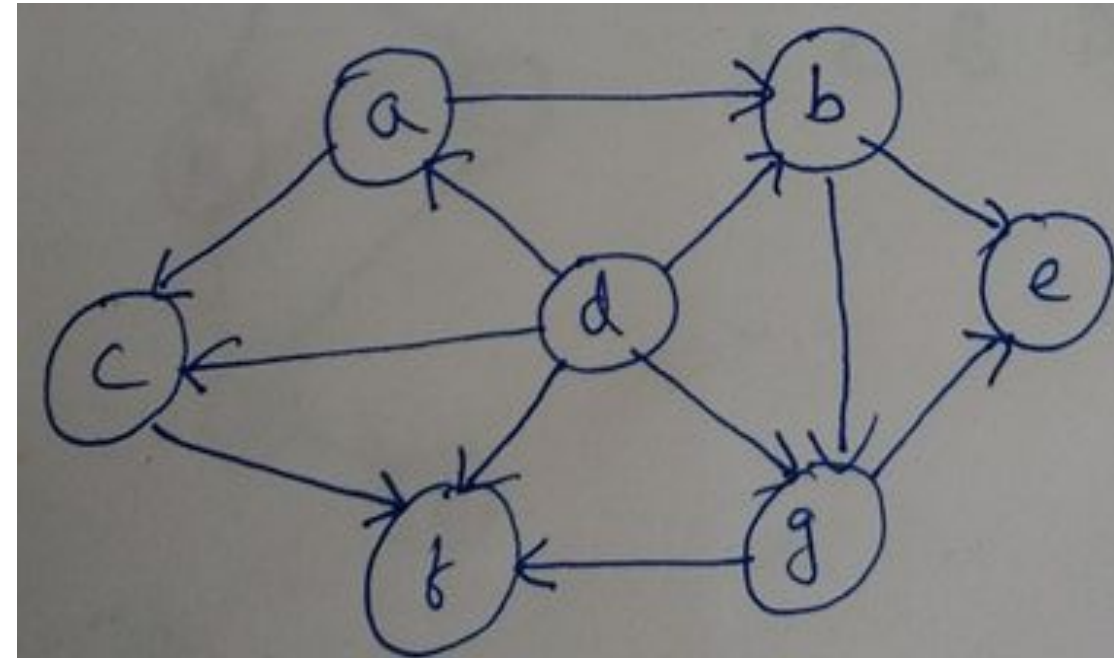
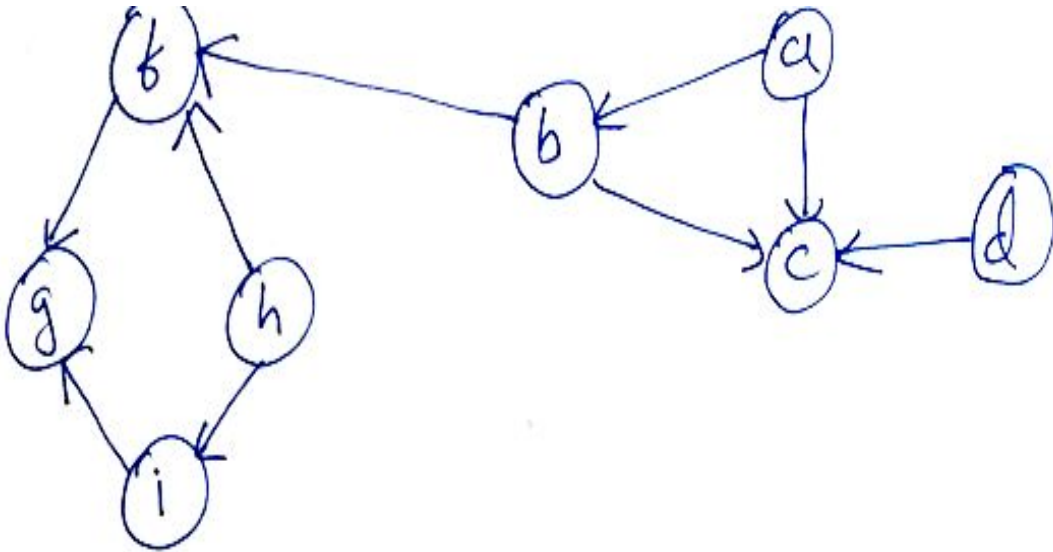
Answer: should be a DAG



The topological sorting is impossible.

Homework Problem

- Find topological Sorting order for the following graphs using Source removal methods



-
- AIM: Write program to obtain the Topological ordering of vertices in a given digraph. Note: In the record book students should Source removal method
 - Algorithm `topological_sort(a,n,T)`
 - `//purpose` :To obtain the sequence of jobs to be executed resulting topological order
 - `// Input`:a-adjacency matrix of the given graph n-the number of vertices in the graph
 - `//output`: T-indicates the jobs that are to be executed in the order
 - Step 1:[Obtain indgree of each vertex]
 - For j <- 0 to n-1 do
 - sum<-0
 - for i<- 0 to n-1 do
 - sum<-sum+a[i]
 - end for
 - indegree[i] <- sum
 - end for

-
- AIM: Write program to obtain the Topological ordering of vertices in a given digraph. Note: In the record book students should Source removal method
 - Step 2: [Place the independent jobs which have not been processed on the stack]
 - For <- 0 to n-1 do
 - If(indegree[i]=0) //Place the job on the stack
 - top <- top+1
 - s[top] <- i
 - end if
 - end for

-
- Step 3: [Find the topological sequence]
 - While (top!=1)
 - u <- s[top]
 - top <- top -1
 - Add u to solution vector T
 - For each vertex v adjacent to u
 - Decrement indegree[v] by one
 - If(indegree[v]=0)
 - Top <- top +1
 - s[top] <- v
 - end if
 - end for
 - end while
 - Step 4: write T
 - Step 5: return

Topological Sorting

Depth First search method:

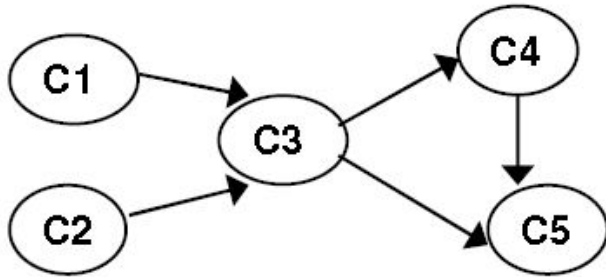
Perform DFS traversal and note down the order in which nodes become dead ends (popped off the stack).

Reversing this order yields a solution to the topological sorting problem.

Provided of course, no back edge has been encountered during traversal. If a back edge has been encountered then the digraph is not DAG (Directed Acyclic Graph) and topological sorting is impossible.

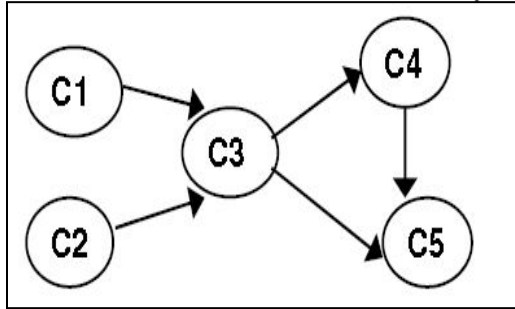
Example

Apply DFS – based algorithm to solve the topological sorting problem for the given graph:



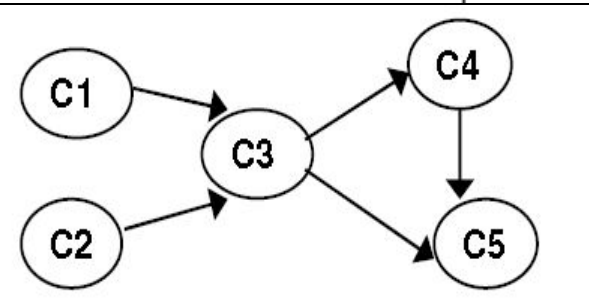
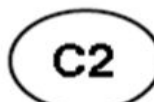
Step	Graph	Remarks
1	<pre>graph LR; C1((C1))</pre>	Insert C1 into stack C1(1)
2	<pre>graph LR; C1((C1)) --> C3((C3))</pre>	Insert C2 into stack C3 (2) C1(1)
3	<pre>graph LR; C1((C1)) --> C3((C3)) --> C4((C4))</pre>	Insert C4 into stack C4 (3) C3 (2) C1(1)
4	<pre>graph LR; C1((C1)) --> C3((C3)) --> C4((C4)) --> C5((C5))</pre>	Insert C5 into stack C5 (4) C4 (3) C3 (2) C1(1)

Example (Contd...)



5	NO unvisited adjacent vertex for C5, backtrack	Delete C5 from stack C5 (4, 1) C4 (3) C3 (2) C1(1)
6	NO unvisited adjacent vertex for C4, backtrack	Delete C4 from stack C5 (4, 1) C4 (3, 2) C3 (2) C1(1)
7	NO unvisited adjacent vertex for C3, backtrack	Delete C3 from stack C5 (4, 1) C4 (3,2) C3 (2, 3) C1(1)
8	NO unvisited adjacent vertex for C1, backtrack	Delete C1 from stack C5 (4, 1) C4 (3,2) C3 (2, 3) C1(1, 4)

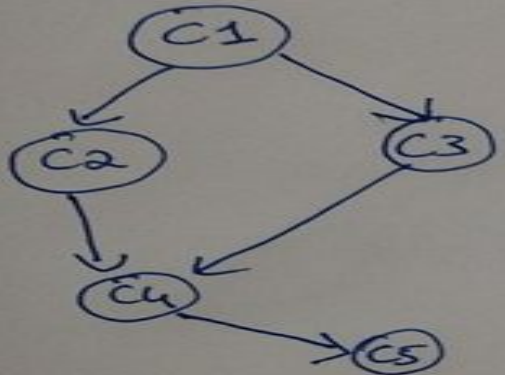
Example (Contd...)

Stack becomes empty, but there is a node which is unvisited, therefore start the DFS again from arbitrarily selecting a unvisited node as source			
	9		Insert C2 into stack C5 (4, 1) C4 (3, 2) C3 (2, 3) C1(1, 4) C2(5)
	10	NO unvisited adjacent vertex for C2, backtrack	Delete C2 from stack C5 (4, 1) C4 (3, 2) C3 (2, 3) C1(1, 4) C2(5, 5)
	Stack becomes empty, NO unvisited node left, therefore algorithm stops. The popping – off order is: C5, C4, C3, C1, C2, Topologically sorted list (reverse of pop order): C2, C1 → C3 → C4 → C5		

Topological Sorting: Using Depth First Search method

Topological Ordering

$C1, C2, C3, C4, C5$
 $C1, C3, C2, C4, C5$ } Two Solutions



DFS method

	C1	C2	C3	C4	C5
Visited	✓	✓	✓	✓	✓
Status	1	1	1	1	1

The order in which nodes are visited
Output: C1 C2 C4 C5 C3

Adjacent vertex

C1: C2, C3
C2: C4
C4: C5
C5:
C3: C4

C5
C4
C2 C3
C1

Stack

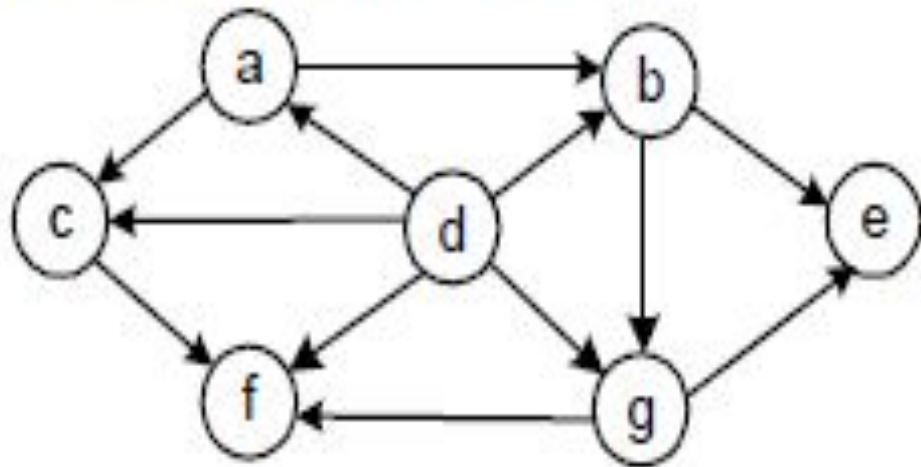
The order in which nodes become Dead ends (or popped off)

Topological order

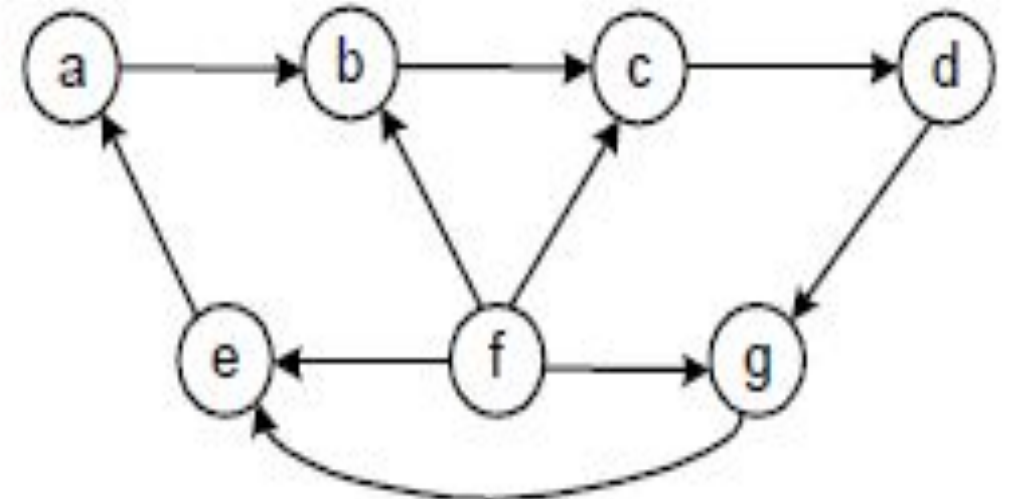
C5
C4
C2
C3
C1

Homework Problem

Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



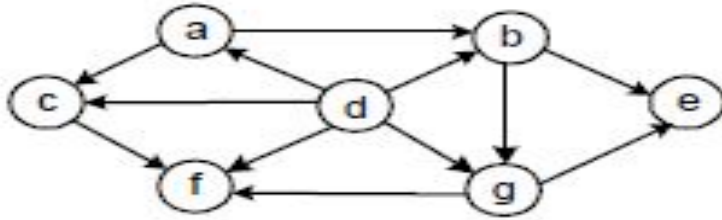
(a)



(b)

Answer

a. The digraph and the stack of its DFS traversal that starts at vertex a are given below:



$$\begin{array}{cccc} & f & & \\ e & g & & \\ b & & c & \\ a & & & d \end{array}$$

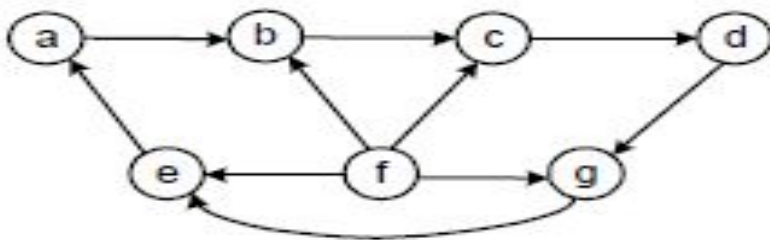
The vertices are popped off the stack in the following order:

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

The topological sorting order obtained by reversing the list above is

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

b. The digraph below is not a dag. Its DFS traversal that starts at a encounters a back edge from e to a :



$$\begin{array}{c} e \\ g \\ d \\ c \\ b \\ a \end{array}$$

Homework Problem

- For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?

Answer:

- For a digraph with n vertices and no edges, any permutation of its vertices solves the topological sorting problem. Hence, the answer to the question is $n!$.

Algorithms for Generating Combinatorial Objects

The most important types of combinatorial objects are ***permutations, combinations, and subsets*** of a given set.

They typically arise in problems that require a consideration of different choices.

Generating Permutations

- Travelling Salesman Problem – all permutations of the cities.
- We assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n ; more generally, they can be interpreted as indices of elements in an n -element set $\{a_1, \dots, a_n\}$.
- **The decrease-by-one technique for the problem of generating all $n!$ permutations of $\{1, \dots, n\}$, would be $n(n-1)! = n!$**
- We can insert n in the previously generated permutations either left to right or right to left. It turns out that it is beneficial to start with inserting n into $1 \ 2 \ \dots \ (n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1, \dots, n-1\}$ needs to be processed.
- An example of applying this approach **bottom up** for $n = 3$ is given below:
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

Generating Permutations

- It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n .
- It can be done by **associating a direction with each element k in a permutation**. We indicate such a direction by a small arrow written above the element in question, e.g.,

$\overrightarrow{3} \overleftarrow{2} \overrightarrow{4} \overleftarrow{1}.$

- 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. For example, for the permutation

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

- 3 and 4 are mobile while 2 and 1 are not.

- Identify the mobile element/integer--->

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

Generating Permutations

Using the notion of a mobile element, we can give the following description of the Johnson-Trotter algorithm for generating permutations.

ALGORITHM *JohnsonTrotter*(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 $\overset{\leftarrow}{1} \overset{\leftarrow}{2} \dots \overset{\leftarrow}{n}$

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

Johnson and Trotter algorithm

The Johnson and Trotter algorithm doesn't require to store all permutations of size $n-1$ and doesn't require going through all shorter permutations. Instead, it keeps track of the direction of each element of the permutation.

- ❑ Find out the largest mobile integer in a particular sequence. A directed integer is said to be mobile if it is greater than its immediate neighbor in the direction it is looking at.
- ❑ Switch this mobile integer and the adjacent integer to which its direction points.
- ❑ Switch the direction of all the elements whose value is greater than the mobile integer value.
- ❑ Repeat the step 1 until unless there is no mobile integer left in the sequence.

- ❑ <https://www.geeksforgeeks.org/johnson-trotter-algorithm/>

Generating Permutations

- Here is an application of this algorithm for $n = 3$, with the largest mobile element shown in bold:

←←← ←←← ←←← →←← ←→← ←←→
1 2 **3** 1 **3** 2 3 1 **2** **3** 2 1 2 **3** 1 2 1 **3**.

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

- 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 **$\Theta(n!)$, which is horribly slow for all but very small values of n .**

Generating Permutations

Generate all the permutations for $n = 4$: 1 2 3 4

□	1234	1243	1423	4123
□	4132	1432	1342	1324
□	3124	3142	3412	4312
□	4321	3421	3241	3214
□	2314	2341	2431	4231
□	4213	2413	2143	2134

← ← ← ←
1 2 3 4

← ← ← ←
1 2 4 3

← ← ← ←
1 4 2 3

← ← ← ←
4 1 2 3

→ ← ← ←
4 1 3 2

← → ← ←
1 4 3 2

← ← → ←
1 3 4 2

← ← ← →
1 3 2 4

← ← ← ←
3 1 2 4

← ← ← ←
3 1 4 2

← ← ← ←
3 4 1 2

← ← ← ←
4 3 1 2

→ → ← ←
4 3 2 1

→ → ← ←
3 4 2 1

→ ← → ←
3 2 4 1

→ ← ← →
3 2 1 4

← → ← ←
2 3 1 4

← → ← ←
2 3 4 1

← ← → ←
2 4 3 1

← ← → ←
4 2 3 1

→ ← ← →
4 2 1 3

← → ← →
2 4 1 3

← ← → →
2 1 4 3

← ← → →
2 1 3 4

Lexicographic Permutation

$\overleftarrow{1} \overleftarrow{2} \overleftarrow{3}$ $\overleftarrow{1} \overleftarrow{3} \overleftarrow{2}$ $\overleftarrow{3} \overleftarrow{1} \overleftarrow{2}$ $\overrightarrow{3} \overleftarrow{2} \overleftarrow{1}$ $\overleftarrow{2} \overrightarrow{3} \overleftarrow{1}$ $\overleftarrow{2} \overleftarrow{1} \overrightarrow{3}$.

□ 123 132 213 231 312 321

Lexicographic Permutation

- ALGORITHM LexicographicPermute(n)
- //Generates permutations in lexicographic order
- //Input: A positive integer n
- //Output: A list of all permutations of $\{1, \dots, n\}$ in lexicographic order
- initialize the first permutation with $12 \dots n$
- while last permutation has two consecutive elements in increasing order do
- let i be its largest index such that $a_i < a_{i+1}$ // $a_{i+1} > a_{i+2} > \dots > a_n$
- 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 // $a_{i+1}a_{i+2} \dots a_n$ 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 Subsets

- Knapsack problem –all subsets of the items.
- The decrease-by-one idea is immediately applicable to this problem, too. 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

- A convenient way of solving the problem directly 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.
- For 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., $\{a_1, a_2, a_3\}$, and 110 will represent $\{a_1, a_2\}$.
- With this correspondence in place, we can generate all the bit strings of length n by generating successive binary numbers from 0 to $2^n - 1$, padded, when necessary, with an appropriate number of leading 0's.
- For example, for the case of $n = 3$, we obtain

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

Generating Subsets

- We require a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit.
- In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element.
- The solution is the **binary reflected Gray code**.
- For example, for $n = 3$, we can get
000 001 011 010 110 111 101 100

Generating Subsets

ALGORITHM $BRGC(n)$

//Generates recursively the binary reflected Gray code of order n

//Input: A positive integer n

//Output: A list of all bit strings of length n composing the Gray code

if $n = 1$ make list L containing bit strings 0 and 1 in this order

else generate list $L1$ of bit strings of size $n - 1$ by calling $BRGC(n - 1)$

 copy list $L1$ to list $L2$ in reversed order

 add 0 in front of each bit string in list $L1$

 add 1 in front of each bit string in list $L2$

 append $L2$ to $L1$ to get list L

return L

Decrease by a Constant Factor

- Binary Search

Thanks for Listening
