

Decrease and Conquer

Unit 3

Decrease and Conquer – The IDEA

- The Decrease and Conquer algorithm design strategy is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- The general methodology used by this design technique is as follows:
 1. Reduce problem instance to smaller instance of the same problem
 2. Solve smaller instance
 3. Extend solution of smaller instance to obtain solution to original instance

Decrease and Conquer – The IDEA

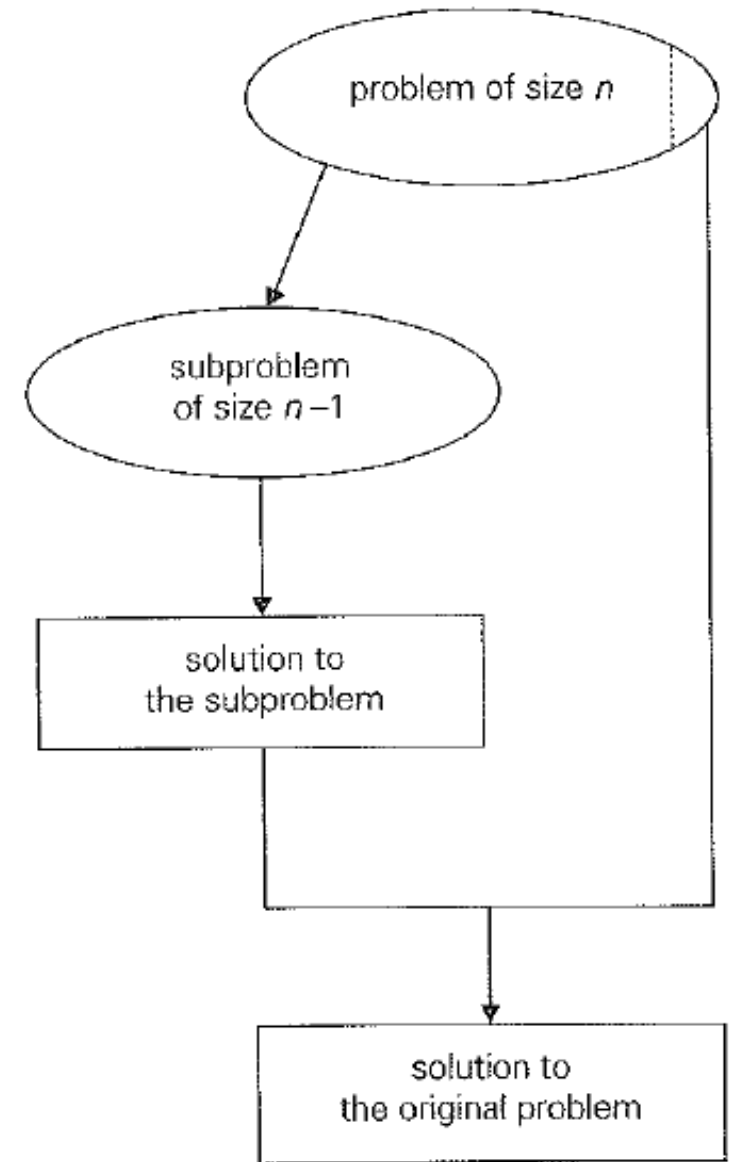
- The solution to a problem using this design strategy can be implemented either bottom – up or top – down.
- This technique is also referred to as *Inductive Approach* or *Incremental Approach*.

Variations of Decrease and Conquer

- Decrease by a constant
- Decrease by a constant factor
- Variable Size Decrease

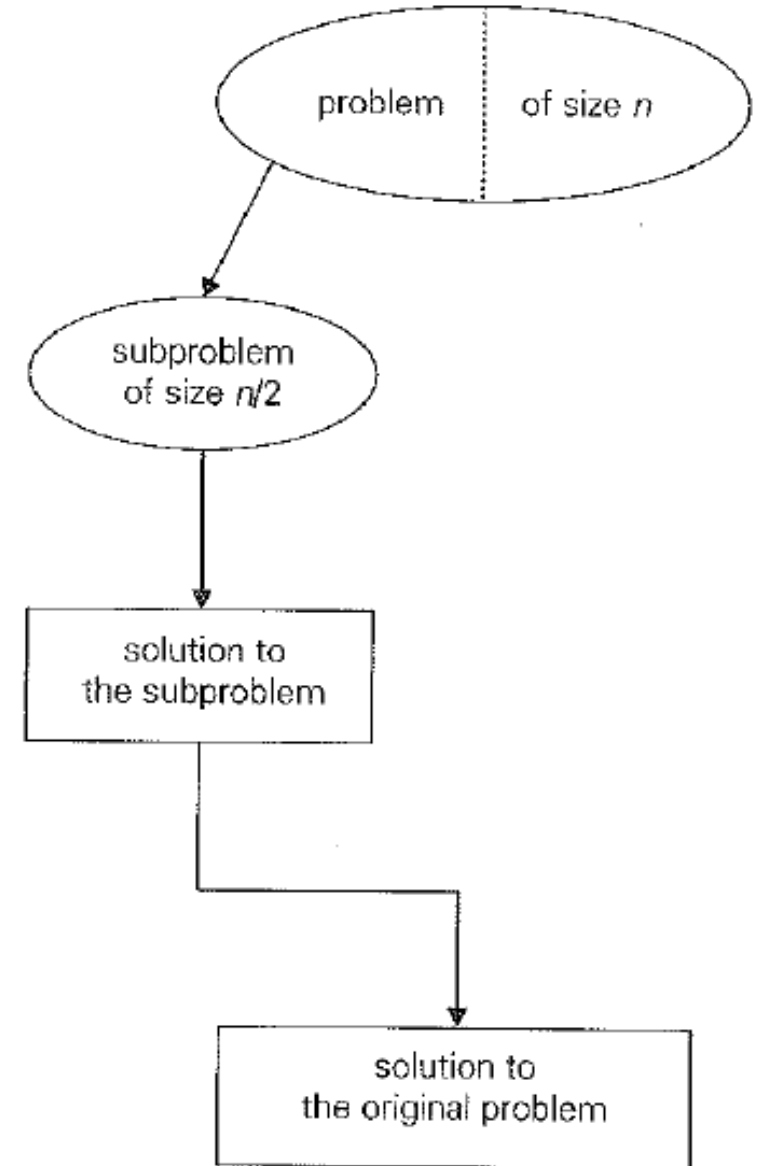
Decrease by a constant

- In this 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 although other constant size reductions do happen.
- Example: $a^n = a^{n-1} * a$



Decrease by a constant factor

- In this variation, the size of an instance is reduced by the same constant factor on each iteration of the algorithm.
- In most cases, this constant factor is equal to two.
- Example: $a^n = (a^{n/2})^2$



Variable Size Decrease

- In this variation, the size reduction pattern varies from one iteration of the algorithm to another.
- Example: Euclid's Algorithm for computing the GCD.
- $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$

INSERTION SORT

Insertion Sort – The IDEA

- It is an example of the Decrease – by – One technique to sorting an array $A[0..n-1]$.
- We assume that the smaller problem of sorting an array of size, $A[0..n-2]$ has already been solved to give us a sorted array of size $n-1$:
 $A[0] \leq A[1] \dots \leq A[n-2]$.
- Then we find an appropriate position for $A[n-1]$ among the sorted elements and insert it there.

Insertion Sort – The 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** $n - 1$ **do**

$v \leftarrow A[i]$

$j \leftarrow i - 1$

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

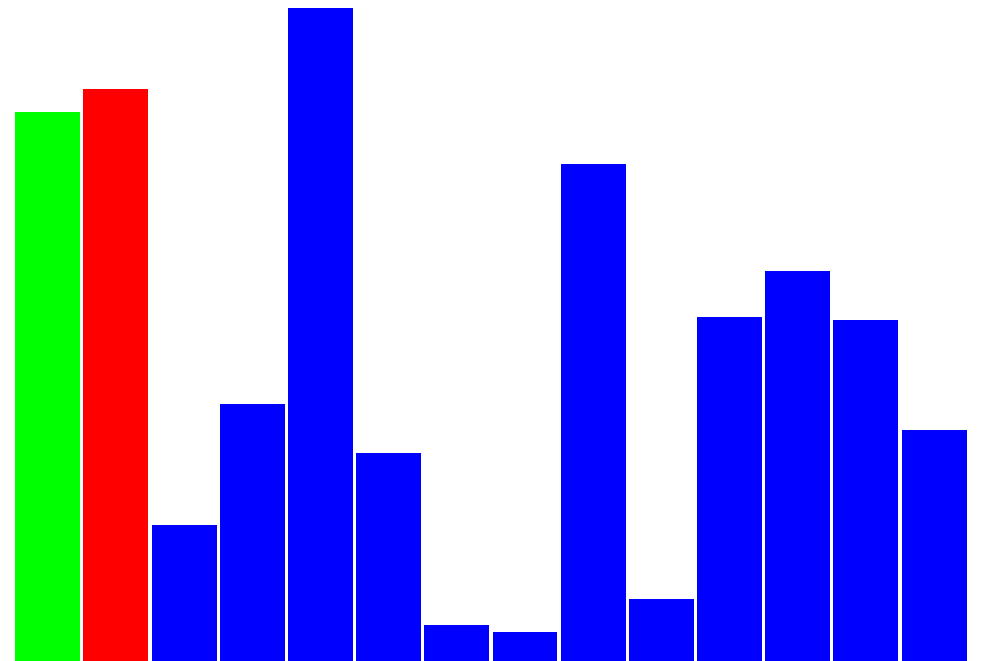
$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

Insertion Sort – An Example

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



Insertion Sort – Analysis

In the worst case:

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

In the best case:

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

Insertion Sort – Analysis

In the average case:

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

DEPTH FIRST SEARCH

Depth First Search – The IDEA

- Visits graph's vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
- Recursive or it uses a stack
 - a vertex is pushed onto the stack when it's reached for the first time
 - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex
- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)

Depth First Search – The IDEA

- It is useful to accompany a DFS traversal by constructing the ***Depth First Search Forest***.
- The starting vertex of the traversal serves as the root of the first tree in such a forest.
- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest.
- The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor. Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest.

Depth First Search – The ALGORITHM

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)

Depth First Search – The ALGORITHM

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*

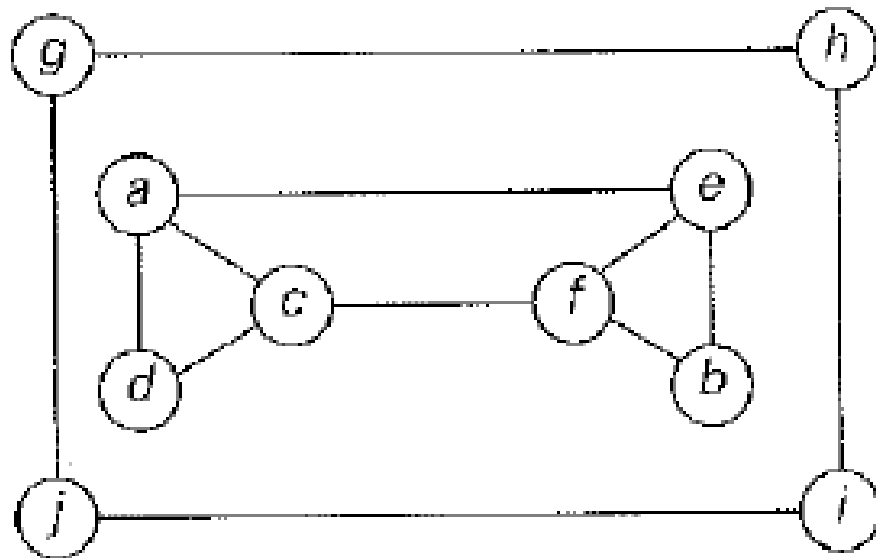
count \leftarrow *count* + 1; mark v with *count*

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

if w is marked with 0

dfs(w)

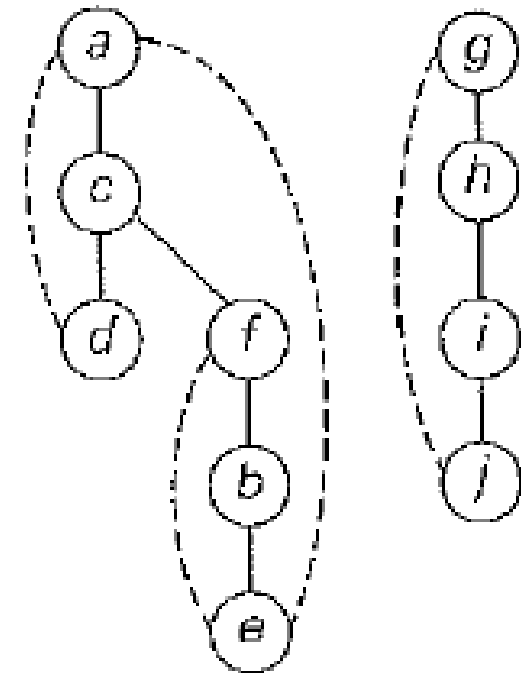
Depth First Search – An Example



(a)

$d_{3,1}$
 $c_{2,5}$
 $a_{1,6}$
 $e_{6,2}$
 $b_{5,3}$
 $f_{4,4}$
 $j_{10,7}$
 $i_{9,8}$
 $h_{8,9}$
 $g_{7,10}$

(b)



(c)

- (a) Graph
- (b) Stack
- (c) Forest

Depth First Search – Analysis

- DFS can be implemented using Adjacency Lists and Adjacency Matrices.
- The efficiency is as follows:
 - adjacency matrices: $\Theta(|V|^2)$.
 - adjacency lists: $\Theta(|V| + |E|)$.

BREADTH FIRST SEARCH

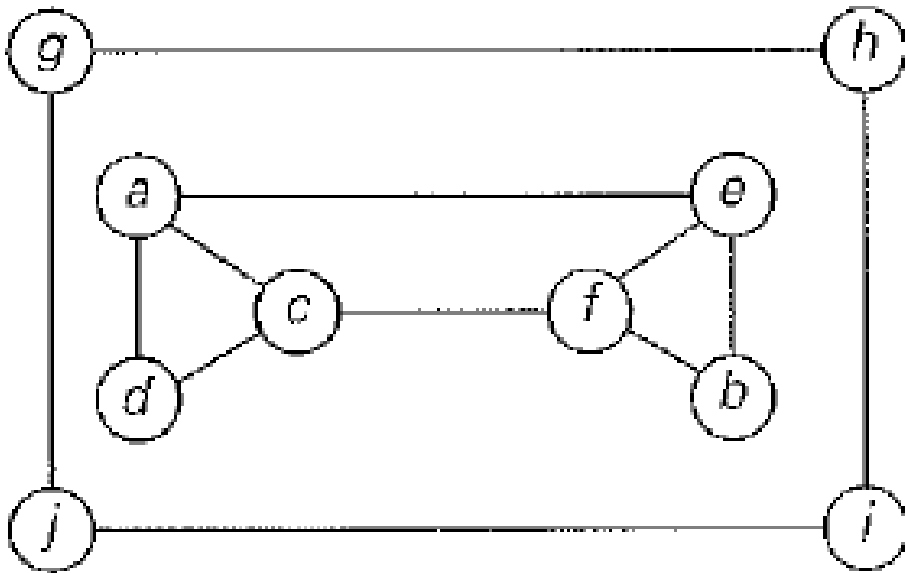
Breadth First Search – The IDEA

- Visits graph vertices by moving across to all the neighbors of the last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Breadth First Search – The IDEA

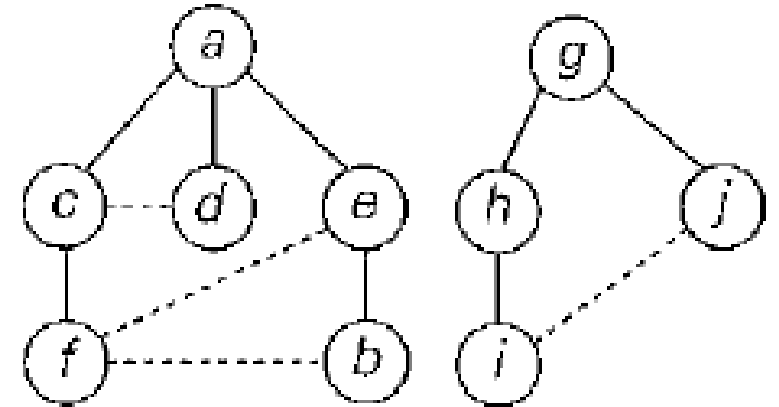
- It is useful to accompany a BFS traversal by constructing the so-called ***breadth-first search forest***.
- The traversal's starting vertex serves as the root of the first tree in such a forest.
- Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***.
- If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***.

Breadth First Search – An Example



(a)

$a_1 c_2 d_3 e_4 f_5 b_6$
 $g_7 h_8 j_9 i_{10}$



(c)

- (a) Graph
- (b) Stack
- (c) Forest

Breadth First Search – The ALGORITHM

ALGORITHM *BFS(G)*

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

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

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

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

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

count $\leftarrow 0$

for each vertex v in V **do**

if v is marked with 0

bfs(v)

Breadth First Search – The ALGORITHM

bfs(v)

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

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

//via global variable *count*

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

while the queue is not empty **do**

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

if w is marked with 0

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

 add w to the queue

 remove the front vertex from the queue

Breadth First Search – Analysis

- BFS can be implemented using Adjacency Lists and Adjacency Matrices.
- The efficiency is as follows:
 - adjacency matrices: $\Theta(|V|^2)$.
 - adjacency lists: $\Theta(|V| + |E|)$.

BFS has the same efficiency as DFS.

BFS Vs DFS

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

Depth First Search – Applications

- Checking connectivity, finding connected components
- Checking acyclicity (if no back edges)
- Finding articulation points and biconnected components
- Searching the state-space of problems for solutions (in AI)

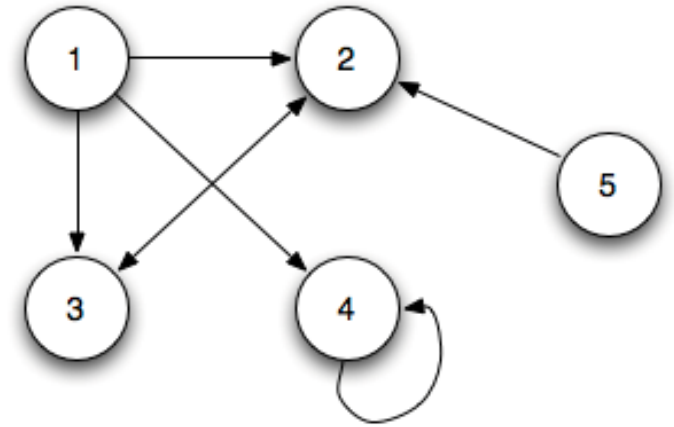
Breadth First Search – Applications

- Checking connectivity, finding connected components
- Checking acyclicity
- Searching the state-space of problems for solutions (in AI)
- can also find paths from a vertex to all other vertices with the smallest number of edges

TOPOLOGICAL SORTING

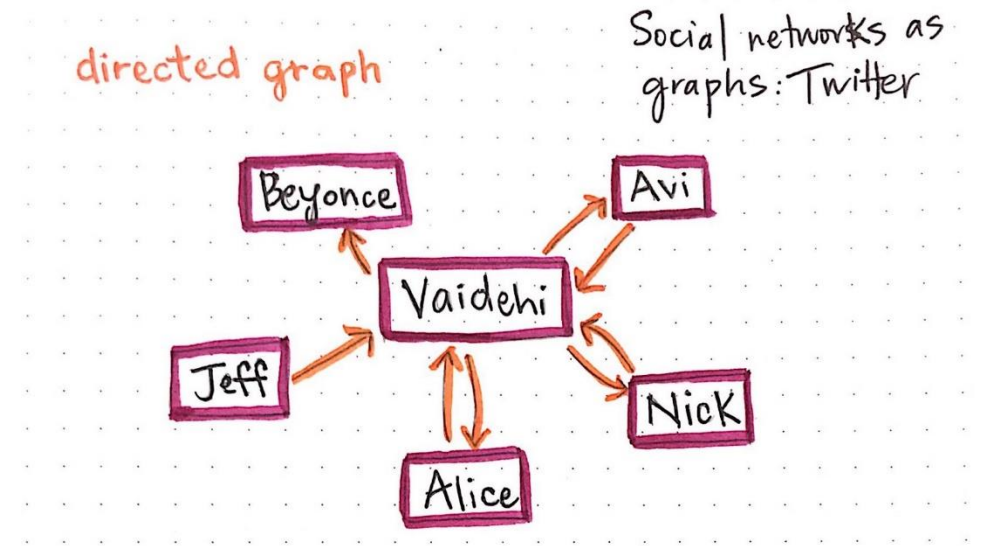
Facts about Directed Graphs

- A *Directed Graph* or *digraph* is a graph with directions specified for its edges.
- The edges indicate a *one-way* relationship, in that each edge can only be traversed in a single direction.
- The *adjacency matrix* and the *adjacency list* are the two principal ways of representing a digraph as well.
- Differences between directed graphs and undirected graphs:
 - The adjacency matrix does not have to be symmetric for a digraph.
 - An edge in a digraph has just one corresponding node in the adjacency list



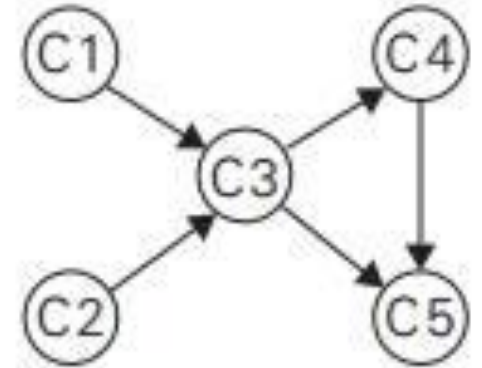
Directed Acyclic Graph

- Twitter can be represented as a directed graph.
- Person A can follow Person B but this need not be vice – versa.
- A *directed cycle* in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex.
- Example: Vaidehi → Alice → Vaidehi
- A directed graph with no cycles is a *Directed Acyclic Graph (DAG)*.



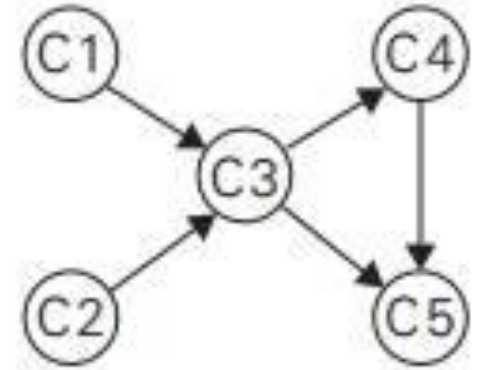
Topological Sorting – The Problem

- Consider a set of five required courses {C1, C2, C3, C4, C5} a part – time student has to take in some degree program.
- The courses can be taken in any order as long as the following pre – requisites are met:
 - C1 and C2 – No Pre – Requisites
 - C3 – Requires C1 and C2
 - C4 – Requires C3
 - C5 – Requires C4 and C3
- The student can take only one course per term.
- In which order should the student take the courses?



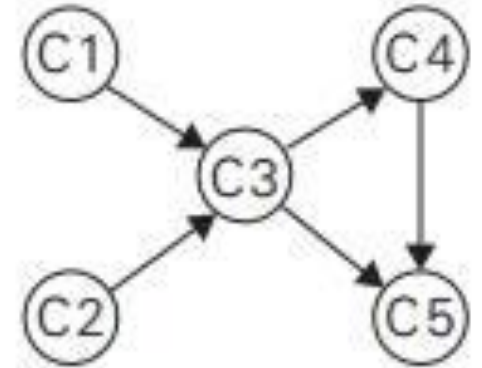
Topological Sorting – The Problem

- If this situation is modeled as a digraph, the problem is *can we list the vertices in such an order that, for every edge in the digraph, the vertex where the edge starts is listed before the vertex where the edge ends?*



Topological Sorting – The Problem

- If this situation is modeled as a digraph, the problem is *can we list the vertices in such an order that, for every edge in the digraph, the vertex where the edge starts is listed before the vertex where the edge ends?*
- This problem cannot have a solution if the graph is a Directed Acyclic Graph.



Topological Sorting – Algorithm 1

- Perform a DFS traversal.
- Note the order in which vertices become dead – ends (popped out of the traversal stack).
- Reversing this order yields a solution to the problem.
- Consider the following example:

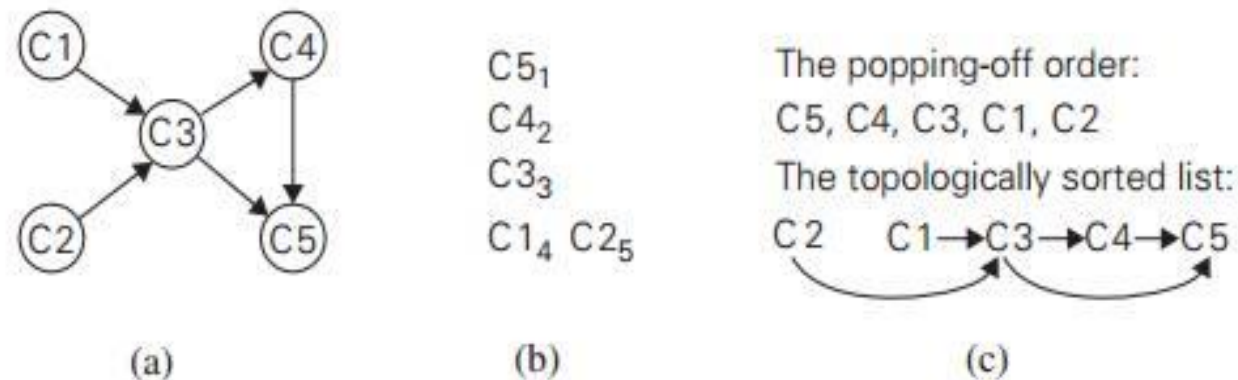


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Topological Sorting – Algorithm 2

- Identify in a remaining digraph a source vertex: A vertex which has no incoming edges.
- Delete this source vertex along with all outgoing edges
- Consider the following example:

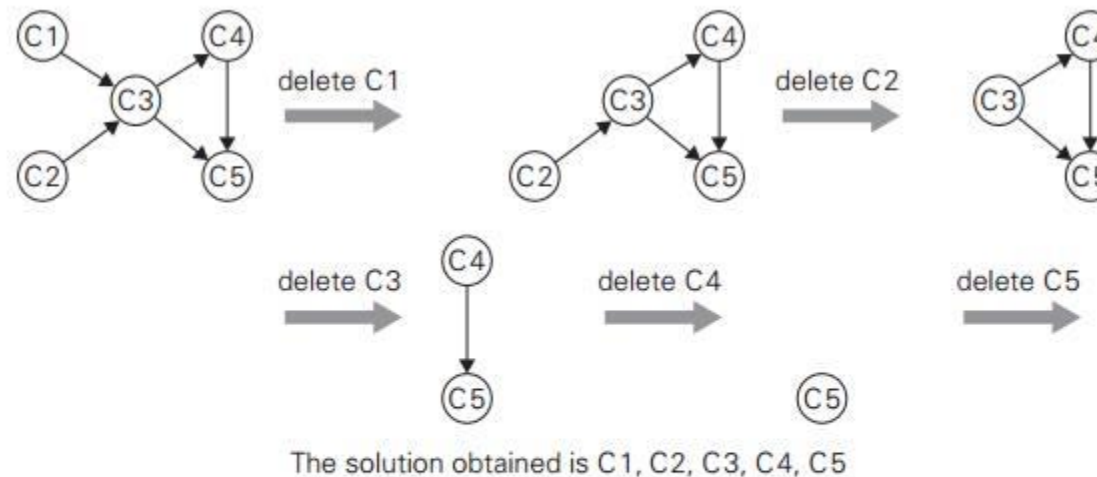


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.



ALGORITHMS FOR GENERATING COMBINATORIAL OBJECTS

GENERATING PERMUTATIONS

Assumption:

- * The set of elements that need to be permuted is simply the set of integers from 1 to n .
- * You can interpret this as the indices of elements in an n – element set $\{a_1, a_2, \dots, a_n\}$.

Question:

What is the *decrease – by – one* technique to solve the problem of generating all $n!$ permutations of the set $\{1, 2, 3, \dots, n\}$?

GENERATING PERMUTATIONS

Answer:


- * Generate all the $(n-1)!$ permutations.
- * Assuming that the smaller problem is solved, the problem of generating $n!$ permutations can be obtained by inserting n in each of the n positions of every permutation of $n - 1$ elements.
- * ***Now, try for $n = 4$.***

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

GENERATING PERMUTATIONS

- * All the permutations generated in this fashion will be distinct. *Why?*
- * The total number of permutations will be $n(n-1)!$ *How?*
- * Continuing the algorithm, n can be inserted in the previously generated permutations starting from either left to right or right to left.
- * It is beneficial to start from right to left and switch direction every time a new permutation of $(n - 1)$ needs to be processed.
- * Following this, each permutation can be obtained from its immediate predecessor just by changing two elements in it. This satisfies the *minimal change* requirement.

GENERATING PERMUTATIONS – JOHNSON TROTTER ALGORITHM

- * It is possible to get the permutations of n elements without explicitly generating the permutations of $n - 1$ elements.
- * This can be done by associating a direction with each element k in a permutation.
- * The direction is indicated by a small arrow written above the element in question.  3 2 4 1.
- * **Mobile Element**: The element k is said to be mobile in such an arrow marked representation if its arrow is pointing to a smaller number adjacent to it.

GENERATING PERMUTATIONS – JOHNSON TROTTER ALGORITHM

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 $\overleftarrow{1} \overleftarrow{2} \dots \overleftarrow{n}$

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

 find its largest mobile element k .

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

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

 add the new permutation to the list

GENERATING PERMUTATIONS – JOHNSON TROTTER ALGORITHM

* Example:

$n = 3$

$\begin{array}{cccccc} \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \rightarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \rightarrow & \leftarrow & \leftarrow & \rightarrow \\ 1 & 2 & \mathbf{3} & 1 & \mathbf{3} & 2 & 3 & 1 & \mathbf{2} & \mathbf{3} & 2 & 1 & 2 & \mathbf{3} & 1 & 2 & 1 & \mathbf{3} \end{array}$

GENERATING PERMUTATIONS – JOHNSON TROTTER ALGORITHM

- * This algorithm is one of the most efficient for generating permutations.
- * Its runtime belongs to $\Theta(n!)$.
- * It is very slow except for very small values of n .
- * But this is the fault of the problem.

GENERATING PERMUTATIONS – LEXICOGRAPHIC ORDER

- * The permutations generated by Johnson – Trotter algorithm are not in lexicographic order.
- * The order in which they would appear in a dictionary if they were interpreted as letters of an alphabet.
- * So, what does this algorithm say?

GENERATING PERMUTATIONS – LEXICOGRAPHIC ORDER

ALGORITHM LexicographicPermute(n)

initialize the first permutation with 12... 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 from a_{i+1} to a_n inclusive

 add the new permutation to the list

GENERATING SUBSETS

THE KNAPSACK PROBLEM

Problem:

Find the most valuable subset of a given set of items which fits in a knapsack with a particular capacity.

Solution:

Generate all subsets of the given set of items, determine the value of each subset and pick the most valuable subset.

This section discusses algorithms for generating all the subsets of a given set.

THE DECREASE – BY – ONE STRATEGY

All the subsets of set $A = \{a_1, a_2, a_3, \dots, a_n\}$

=

All the subsets of the set $\{a_1, a_2, a_3, \dots, a_{n-1}\}$ + sets obtained by putting a_n in each one of the subsets of $\{a_1, a_2, a_3, \dots, a_{n-1}\}$

n	Subsets							
0	ϕ							
1	ϕ	$\{a_1\}$						
2	ϕ	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	ϕ	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

AN ALTERNATIVE SOLUTION

The subset generation problem can be solved by establishing correspondence between subsets of an n – element set $\{a_1, a_2, a_3, \dots, a_n\}$ and all 2^n bit strings of length n .

Thus, if every bit string of length n is generated, it solves the subset generation problem.

Bit Strings	000	001	010	011	100	101	110	111
Subsets	ϕ	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

BINARY REFLECTED GRAY CODE

Is there an algorithm which generates bit strings with minimal change in the bits? That is, from one bit string to the next there is change in just one bit?

000 001 011 010 110 111 101 100

The above sequence of code is called the *Binary Reflected Gray Code*.

BINARY REFLECTED GRAY CODE

The algorithm to generate Binary Reflected Gray Code is as follows:

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

1. Let the list of (n-1)-bit Gray codes be L1. Create another list L2 which is reverse of L1.
2. Modify the list L1 by prefixing a '0' in all codes of L1.
3. Modify the list L2 by prefixing a '1' in all codes of L2.
4. Concatenate L1 and L2. The concatenated list is required list of n-bit Gray codes

BINARY REFLECTED GRAY CODE – A Pinch of History

Frank Gray, after which the code is named, was a researcher in the AT & T Bell Labs. This code was reinvented to minimize the effect of errors in transmitting signals.

Seventy years earlier, this was used by French Engineer Emile Baudot in telegraphy.

received signal.

The binary code with which the present invention deals may take various forms, all of which have the property that the symbol (or pulse group) representing each number (or signal amplitude) differs from the ones representing the next lower and the next higher number (or signal amplitude) in only one digit (or pulse position). Because this code in its primary form may be built up from the conventional binary code by a sort of reflection process and because other forms may in turn be built up from the primary form in similar fashion, the code in question, which has as yet no recognized name, is designated in this specification and in the claims as the “reflected binary code.”

If, at a receiver station, reflected binary code

Picture of the patent which introduces the term.

DECREASE BY A CONSTANT
FACTOR

FAKE – COIN PROBLEM

THE PROBLEM

Among n identical looking coins, one is fake. The fake coin is known to be lighter than a genuine one. How do you identify the fake coin?



THE SOLUTION

The solution to the problem involves dividing the n coins into two piles of $\text{floor}(n/2)$ each. If there are odd number of coins, one is left outside.

If the piles weigh the same, the coin put aside is the fake coin.

Otherwise, we can proceed in the same manner with the lighter pile, which must contain the fake coin.

The number of weighings $W(n)$ needed by the algorithm in the worst case:

Recurrence: $W(n) = W(\text{floor}(n/2)) + 1$ for $n > 1$, $W(1) = 0$

THE SOLUTION

This recurrence is similar to the recurrence set up by Binary Search.

Thus, the algorithm takes $\lfloor \log_2 n \rfloor$ number of weighings.

RUSSIAN PEASANT MULTIPLICATION

THE RUSSIAN PEASANT MULTIPLICATION

The Russian Peasant multiplication or multiplication a la russe is a non – orthodox algorithm for multiplying two positive integers n and m .

If the instance size is measured by n , then we reduce the instance into half, i.e, into $n/2$ as follows:

$$n * m = (n/2) * 2m \text{ (when } n \text{ is even)}$$

$$n * m = [(n - 1) * 2m + m] \text{ (when } n \text{ is odd)}$$

THE RUSSIAN PEASANT MULTIPLICATION

The Russian Peasant multiplication or multiplication a la russe is a non – orthodox algorithm for multiplying two positive integers n and m .

If the instance size is measured by n , then we reduce the instance into half, i.e, into $n/2$ as follows:

$$n * m = (n/2) * 2m \text{ (when } n \text{ is even)}$$

$$n * m = [(n - 1) * 2m + m] \text{ (when } n \text{ is odd)}$$

THE RUSSIAN PEASANT MULTIPLICATION - EXAMPLE

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1,040	
1	2,080	(+1040)
	2,080	$+(130 + 1040) = 3,250$

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1,040	1,040
1	2,080	2,080
		<u>3,250</u>

(b)

THE RUSSIAN PEASANT MULTIPLICATION

In figure (a), we can see that the extra addends are in the rows that have odd values in the first column.

Thus in figure (b), we can find the product by simply adding the elements in the m column that have an odd number in the n column.

This method leads to fast hardware implementations since doubling and halving of binary numbers can be achieved by shifts.