Decrease and Conquer

Chapter 5

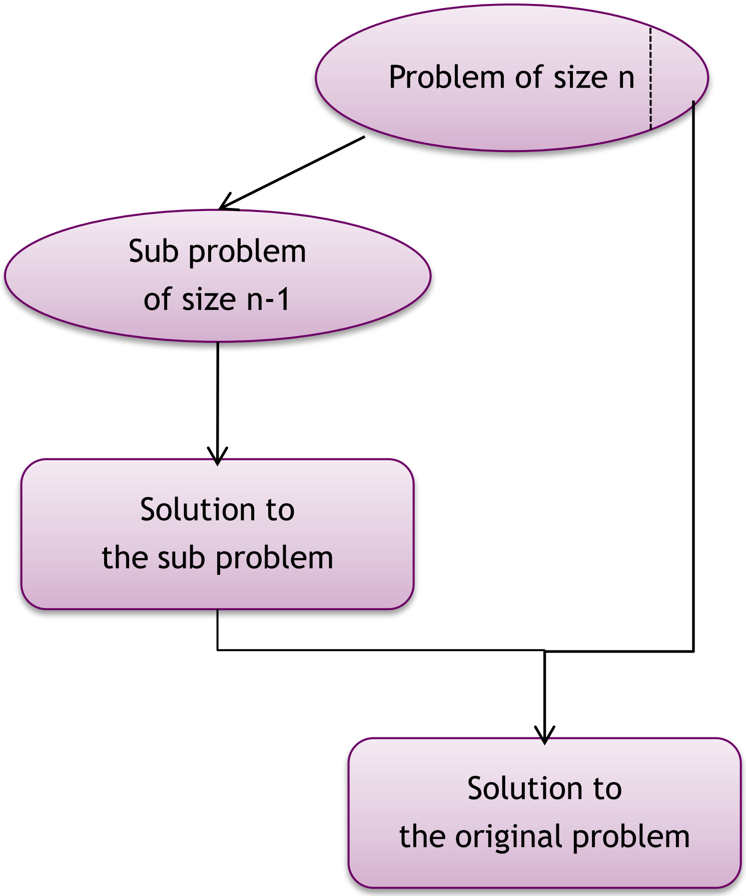
Decrease and Conquer

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 problem

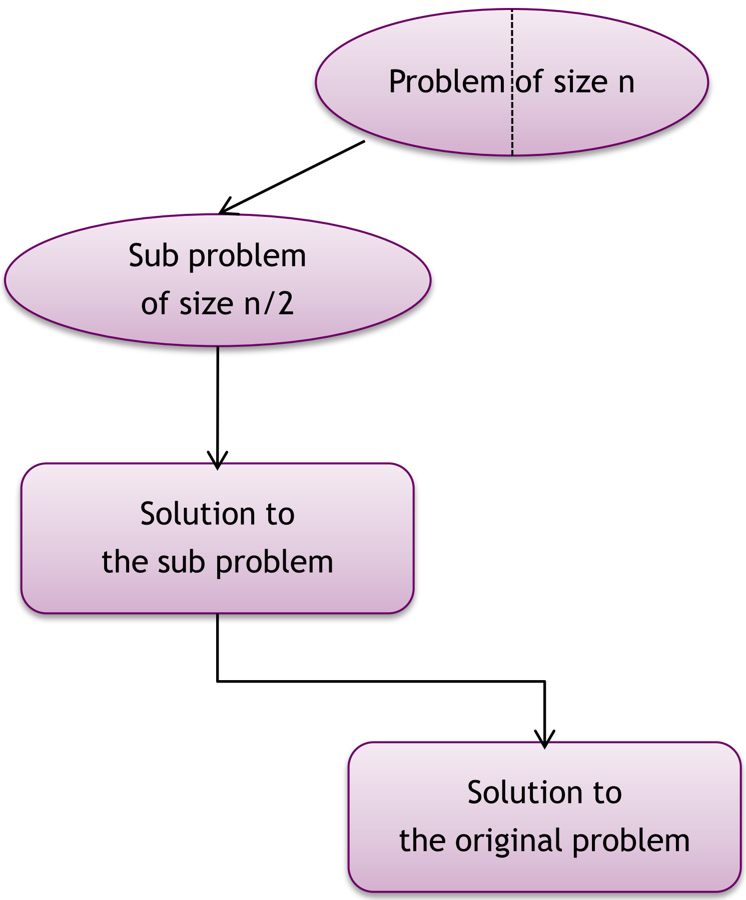
3 Variations of Decrease and Conquer

1. *Decrease by A constant normally one :*
   - Insertion sort
   - DFS
   - BFS
   - Topological sorting
2. *Decrease by a constant factor*
   - Binary search
   - Fake-coin problems
3. *Variable-size decrease*
   - Euclid's algorithm

Decrease (by one) and Conquer Technique



Decrease (by half) and Conquer Technique



Algorithm Insertion Sort

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

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Insertion Sort Example

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

Example of sorting with insertion sort. A vertical bar separates the sorted part of the input from the remaining elements; the element being inserted is in bold.

Efficiency of Insertion Sort

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

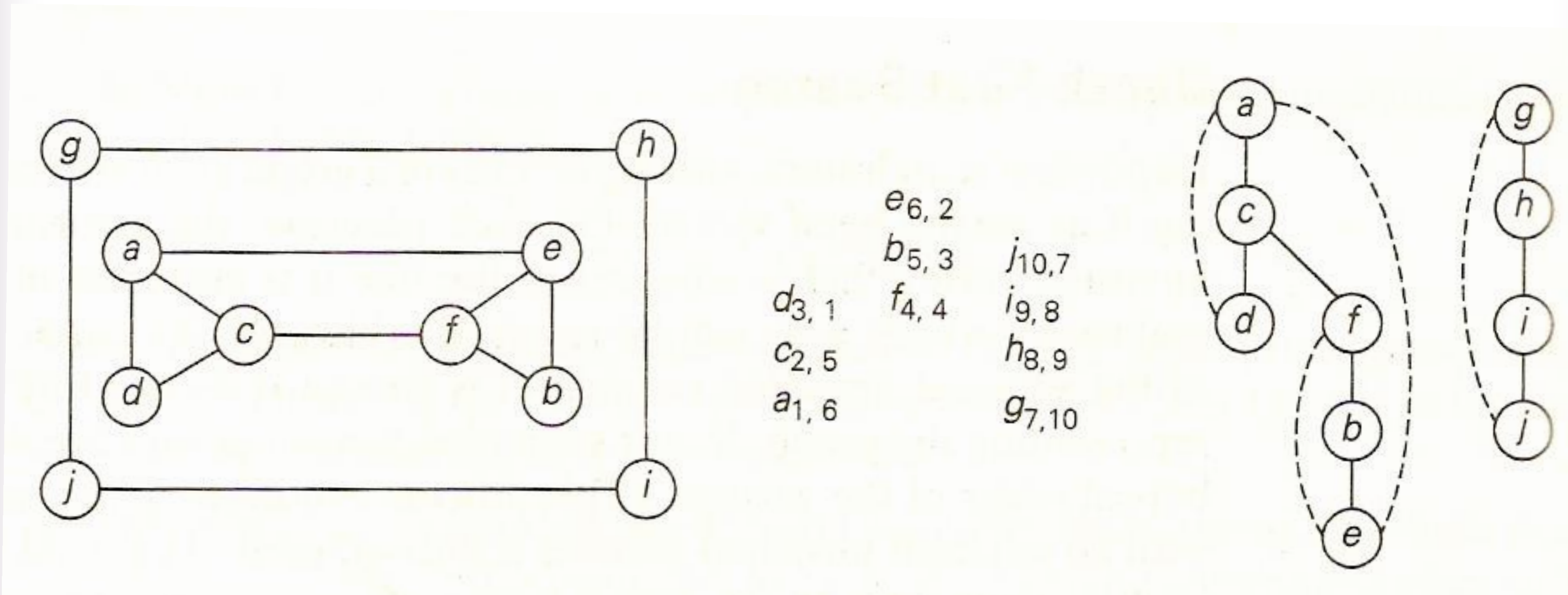
$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

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

Graph Traversal

- Many problems require processing all graph vertices in systematic fashion
- Graph traversal algorithms:
  - Depth-first search
  - Breadth-first search

Depth-first search



(a) (b) (c)

Algorithm DFS(G)

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
        $dfs(v)$

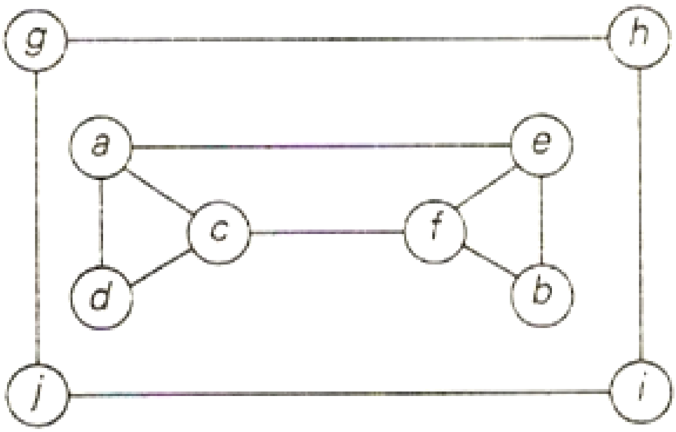
Algorithm dfs(v)

$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex v and
//assigns them the numbers 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
        $dfs(w)$

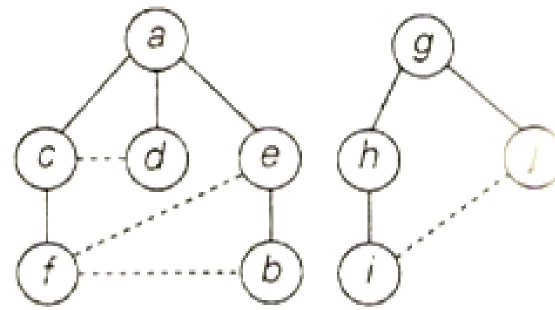
Efficiency of Depth-first search

- Adjacency matrix representation : $\Theta(|V|^2)$
- Adjacency list representation : $\Theta(|V|+|E|)$

Breadth-first search



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



Algorithm BFS(v)

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 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
        $bfs(v)$

Algorithm bfs(v)

```
bfs(v)
//visits all the unvisited vertices connected to vertex v
//and assigns them the numbers 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's vertex v do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove vertex v from the front of the queue
```

Efficiency of Breadth-first search

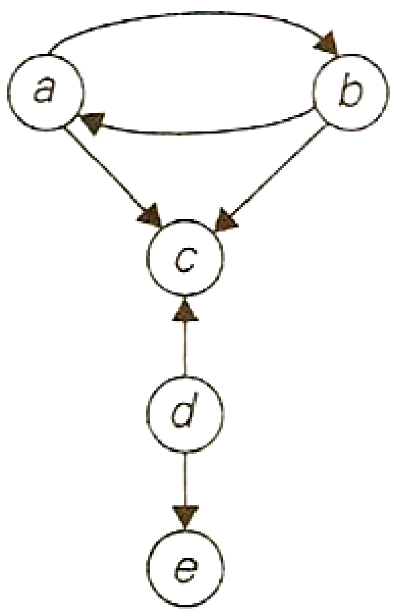
- Adjacency matrices: $\Theta(V^2)$
- Adjacency linked lists: $\Theta(V+E)$

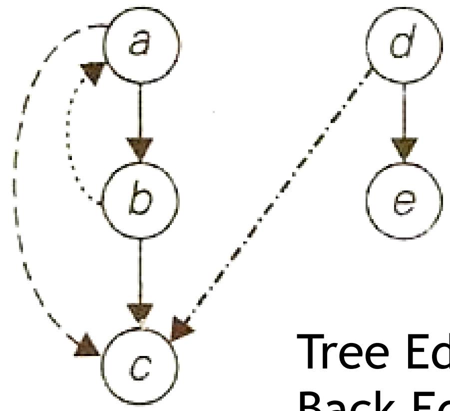
Comparison between DFS & BFS

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 ordering	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	$0(\|V^2\|)$	$0(\|V^2\|)$
Efficiency for adjacent linked lists	$0(\|V\| + \|E\|)$	$0(\|V\| + \|E\|)$

Topological Sorting

- These problem deals with digraphs only.
- We can apply DFS & BFS algorithms for digraphs.

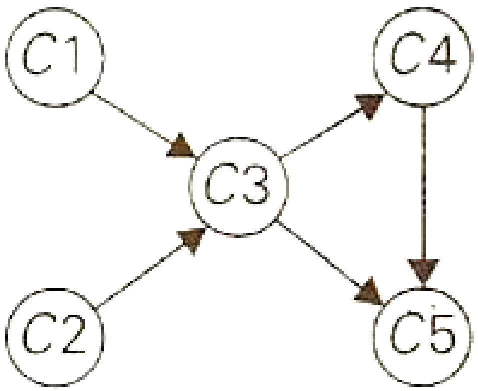




Tree Edges	: ab, bc, de
Back Edges	: ba
Forward Edges	: ac
Cross Edges	: dc

Topological Sorting

- If DFS forest has no back edges then the graph is a dag (directed acyclic graph).
- For topological sorting we can consider only dags.

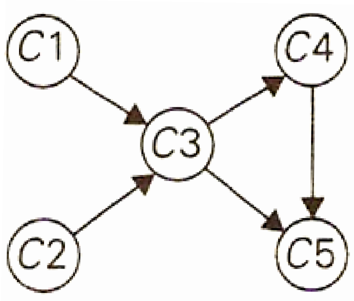


Topological Sorting

- **Definition: Topological sorting for Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.

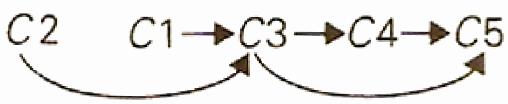
Topological sorting Algorithms

1. Application of DFS:
   - Note the order in which vertices become dead ends.(poped off the stack)
   - Reverse this order to get topological order.



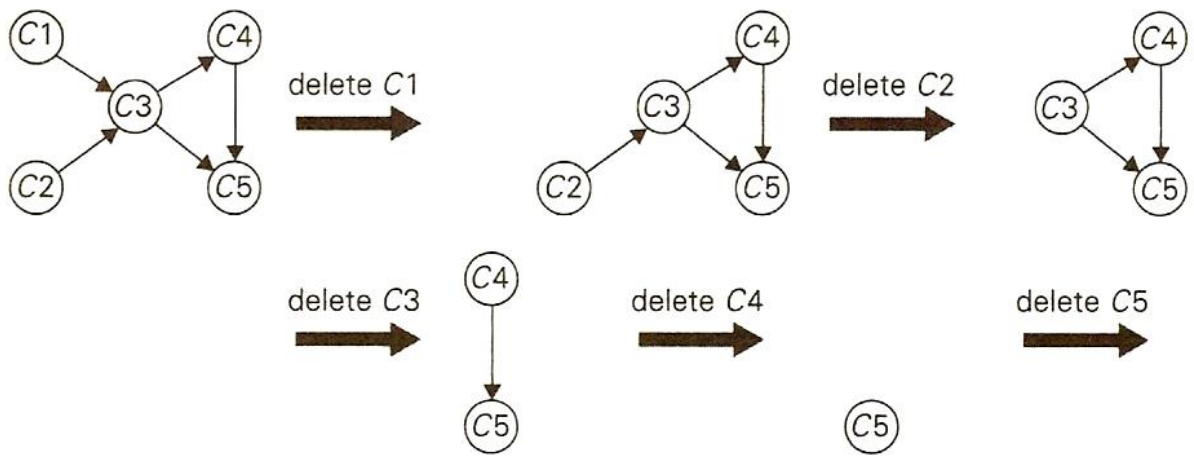
$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:



Topological sorting Algorithms

2. Application of decrease and conquer technique(source removal method)
   - Identify a source vertex whose in-degree is zero. Delete the vertex with all the edges outgoing from it.
   - Repeat the process until all vertices are removed.



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

- Complexity $\Theta(|V|^2)$ using matrix and $\Theta(|V|+|E|)$ using adjacency linked lists

Generating Combinatorial objects

- Combinatorial objects:
  - Permutations
  - Combinations
  - Subsets of a given set

Generating Permutations (3 methods)

1) Minimal change requirement algorithm:

- Here we need to insert n in previously generated permutation.
- First start with right to left and switch the direction every time a new permutation needs to be processed.
- Here each permutation can be obtained from its immediate predecessor by exchanging just 2 elements in it.

start	1
insert 2	$\underbrace{12 \quad 21}_{\text{right to left}}$
insert 3	$\underbrace{123\ 132\ 312}_{\text{right to left}}\ \underbrace{321\ 231\ 213}_{\text{left to right}}$

FIGURE 5.12 Generating permutations bottom up

Generating Permutations

2) using 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 $\{1, \ldots, n\}$
Initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while there exists a mobile integer k **do**
    find the largest mobile integer k
    swap k and the adjacent integer its arrow points to
    reverse the direction of all integers that are larger than k

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

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

3) Lexicographic order permutation

- If $a_{n-1} < a_n$ simply swap last 2 elements.
- If $a_{n-1} > a_n$ & if $a_{n-2} < a_{n-1}$
  - Rearrange last 3 elements.
- Ex: 123 132 213 231 312 321
- General procedure: scan the current permutation from right to left for the first pair of consecutive elements such that $a_i < a_{i+1}$
- Find the smallest digit in the tail larger than a_i and put it in position I
- Positions from i+1 to n are filled in increasing order of remaining elements

Generating Subsets

1. Decrease by one Technique

- If there are n items 2^n sub sets are possible.
- Let $A=\{a_1,a_2,a_3,\ldots.a_n\}$ then all possible subsets of A is P(A).
- Divide A into 2 groups. 1. Containg a_n 2. Without a_n (all sub sets of $\{a_1,a_2,\ldots\ldots.a_{n-1}\}$)
- We can generate P(A) by including a_n to all possible subsets of $\{a_1,a_2,a_3,\ldots.a_{n-1}\}$.

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

Generating Subsets – 2. using Bit String

- Have a 1-1 correspondence between 2^n subsets of n element set $A=\{a_1,a_2,a_3,\ldots.a_n\}$ and 2^n bit strings $b_1,b_2,b_3,\ldots.b_n$ of length n.
- i.e. for the bit string $b_i=1$ if a_i subset else $b_i=0$.
- Ex:
  - 000 null set
  - 010 $\{a_2\}$

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

3. Minimal change algorithm for generating Bit String

- Using gray codes.
- The gray code of order k denoted by G_k defines an ordering among all k-bit binary numbers,
- G_1 is 0 1
- For k>1 G_k $0[G_{k-1}]$, $1[G_{k-1}]^{reverse}$

- 0 1
- 00 01 11 10
- 000 001 011 010 110 111 101 100

The End

Thank You