CSC 411
Design and Analysis of Algorithms

Chapter 4 Decrease-and-Conquer - Part 1

Instructor: Minhee Jun
junm@cua.edu

Decrease-and-Conquer

Exploit the relationship between a solution to a given instance of a problem and a solution to tis smaller instance.

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

- Can be implemented either top-down or bottom-up
  - Top-down: a recursive implementation
  - Bottom-up: implemented iteratively, starting from the smallest instance
- Also referred to as *incremental* approach

Three major variations of Decrease-and-Conquer

- *Decrease by a constant*
  - The size of an instance is recused by the same constant (usually by 1) on each iteration of the algorithm
- *Decrease by a constant factor*
  - reduce a problem instance by the same constant factor (usually by half) on each iteration of the algorithm.
- *Variable size decrease*
  - The size-reduction pattern varies from one iteration of an algorithm to another

What's the difference?

Consider the exponentiation problem of computing a^n.

($a \neq 0$, and a nonnegative integer n)

- Decrease-by-a-constant
  - The size of an instance is recused by the same constant (usually by 1) on each iteration of the algorithm
- Decrease-by-a-constant-factor
  - reduce a problem instance by the same constant factor (usually by half) on each iteration of the algorithm.

What is the concept of each design strategy?

Decrease by a constant

Recursive definition:

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

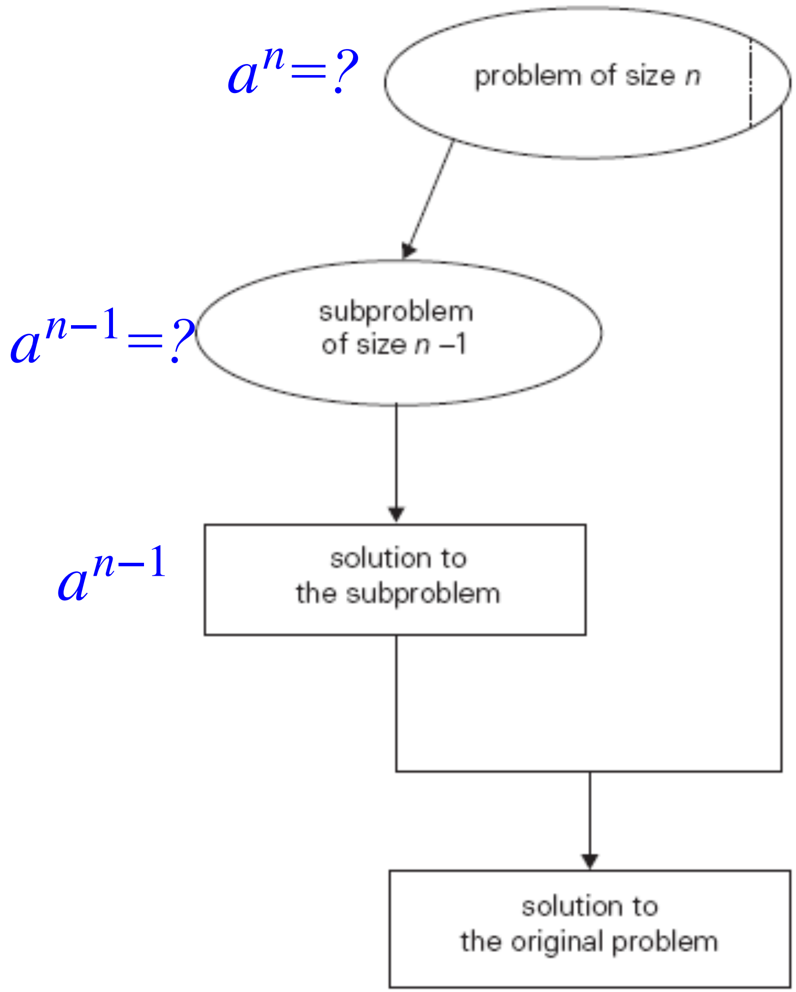


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Decrease by a constant factor

$$a^{n/2} \cdot a^{n/2} = a^n$$

Recursive definition:

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

Efficiency: $\Theta(\log n)$

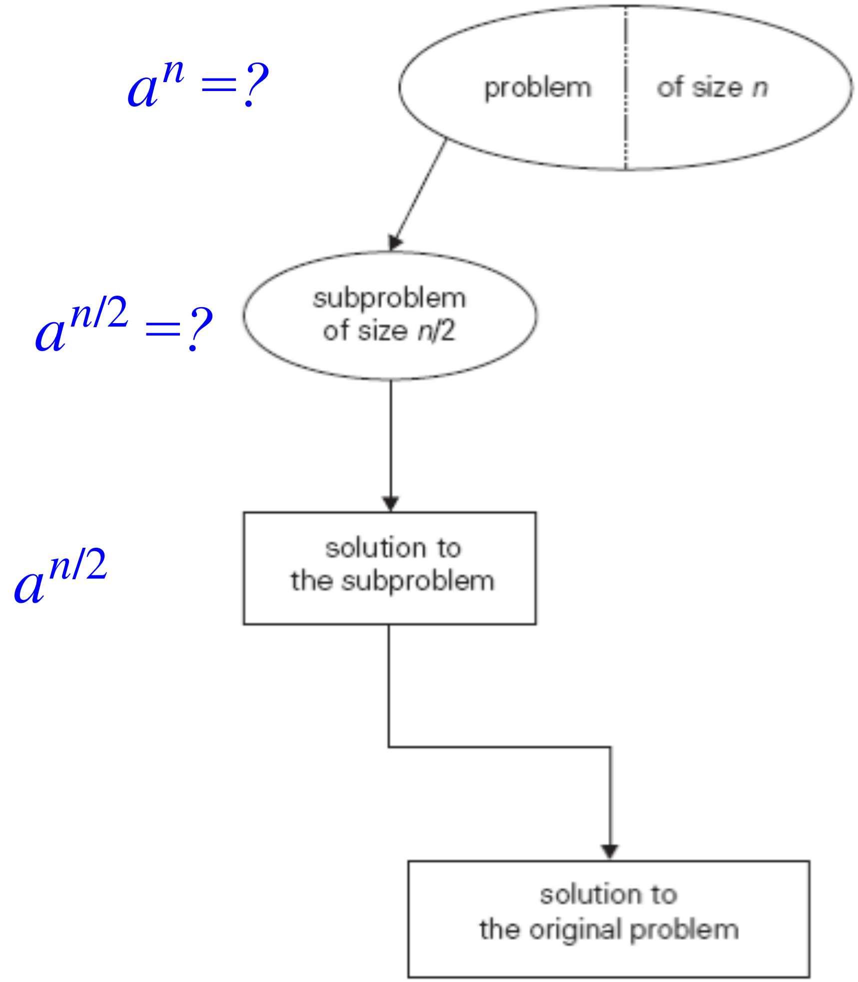


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force:
  - $a^n = a \times a \times a \times a \times \ldots \times a$
- Divide and Conquer:
  - $a^n = a^{n/2} \times a^{n/2}$ (more accurately, $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$)
- Decrease by one:
  - $a^n = a^{n-1} \times a$
  - $f(n) = f(n-1) \times a \quad \text{if } n > 1, \text{ and } f(1) = a$
- Decrease by constant factor:
  - $a^n = (a^{n/2})^2 \quad \text{if } n \text{ is even}$
  - $a^n = (a^{(n-1)/2})^2 \times a \quad \text{if } n \text{ is odd}$

Master Theorem:

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

More of this design strategy will be explained later in chapter 5.

Variable-size decrease

- The size-reduction pattern varies from one iteration of an algorithm to another
- Example: Euclid's algorithm
  - gcd(m,n) = gcd(n, m mod n)

  - The value on the right-hand side is always smaller than the value on the left-hand side.
  - It decreases neither by a constant nor by a constant factor

Three major variations of Decrease-and-Conquer

- *Decrease by a constant* (usually by 1):
  - Graph traversal algorithms (DFS and BFS)
  - Insertion sort
  - Topological sorting
  - Algorithms for generating combinatorial objects
- *Decrease by a constant factor* (usually by half)
  - Binary search
  - Fake-Coin Problem
  - Russian Peasant Multiplication
  - Josephus Problem
- *Variable size decrease*
  - Computing a Median and the Selection Problem
  - Interpolation Search
  - Searching and Insertion in a Binary Search Tree
  - The Game of Nim

4.1 Insertion Sort

- A decrease-by-one technique to sort 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]

Example: Sort 6, 4, 1, 8, 5

$$
\begin{array}{l}
6 \mid \underline{4} \quad 1 \quad 8 \quad 5 \\
4 \quad 6 \mid \underline{1} \quad 8 \quad 5 \\
1 \quad 4 \quad 6 \mid \underline{8} \quad 5 \\
1 \quad 4 \quad 6 \quad 8 \mid \underline{5} \\
1 \quad 4 \quad 5 \quad 6 \quad 8
\end{array}
$$

Insertion Sort: Example

Example: Sort 89, 45, 68, 90, 25, 34, 17 (total n=7)

<table>
<tr><th>Index.</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th></tr>
<tr><td></td><td>89 |</td><td>45</td><td>68</td><td>90</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td></td><td>45</td><td>89 |</td><td>68</td><td>90</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td></td><td>45</td><td>68</td><td>89 |</td><td>90</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td></td><td>45</td><td>68</td><td>89</td><td>90 |</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td></td><td>29</td><td>45</td><td>68</td><td>89</td><td>90 |</td><td>34</td><td>17</td></tr>
<tr><td></td><td>29</td><td>34</td><td>45</td><td>68</td><td>89</td><td>90 |</td><td>17</td></tr>
<tr><td></td><td>17</td><td>29</td><td>34</td><td>45</td><td>68</td><td>89</td><td>90</td></tr>
</table>

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

Insertion Sort: Pseudocode

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 ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

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

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

Insertion Sort: Efficiency

- Time efficiency ?
  - # key comparison ($A[j] > v$) depends on the nature of input.
  - Worst case:

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

  - Best case:

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

  - Average case:

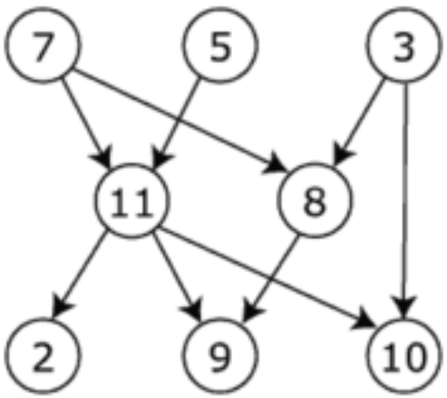
$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Exercise 4.1

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

4.2 Topological Sorting

- Topological sorting algorithms were first studied in the early 1960s in the context of the PERT (program evaluation review technique) for scheduling in project management (Jarnagin 1960).
- The jobs are represented by vertices, and there is an edge from *x* to *y* if job *x* must be completed before job *y* can be started.

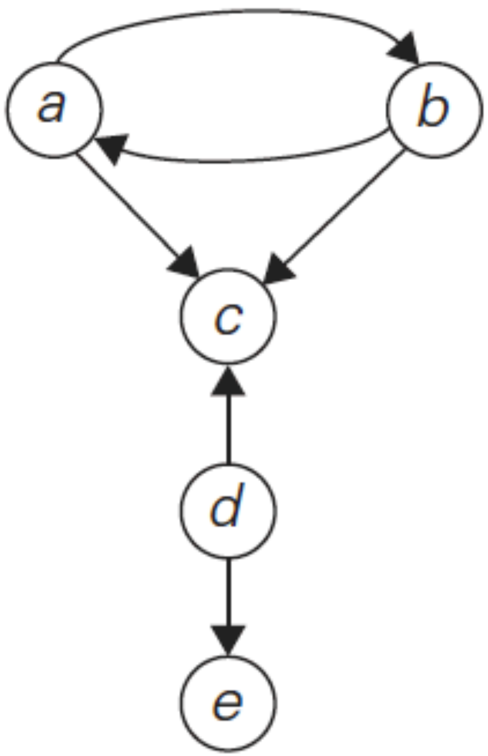


The graph shown to the left has many valid topological sorts, including:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10

Directed Graph (Digraph)

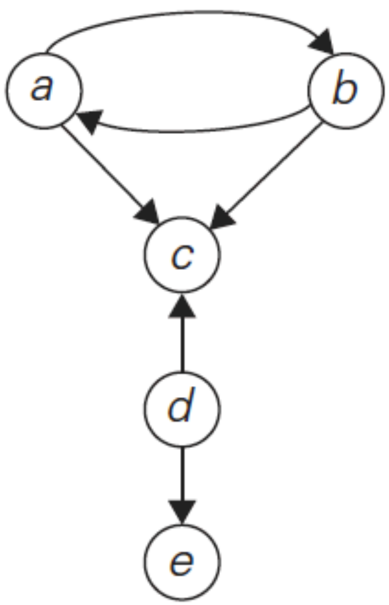
- A directed graph:
  a graph with directions specified for all its edge.



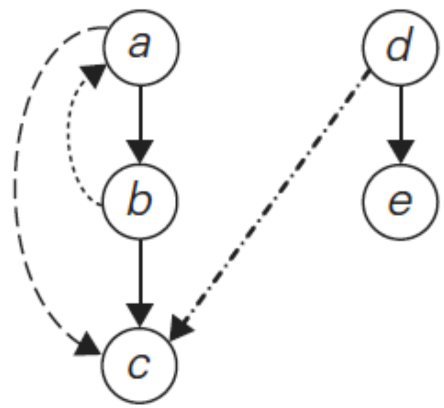
- adjacent matrix:
  not symmetric for digraph
- adjacency list:
  an edge has just one
  corresponding nodes

Directed Graph (Digraph)

- DFS and BFS can be applied to a directed graph:



(a)

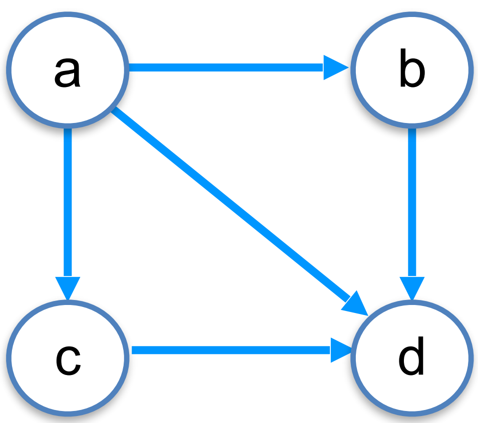


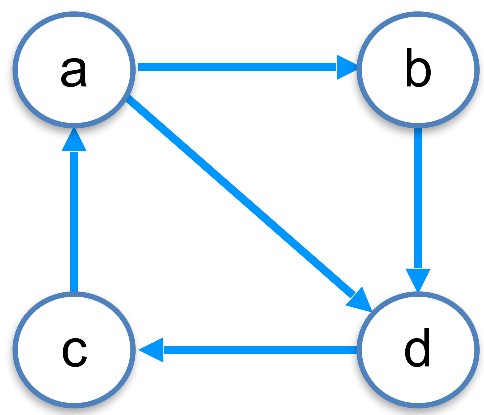
(b)

Directed Acyclic Graph

- A directed graph (digraph):
  a graph with directions specified for all its edge.
  i.e. a directed graph with no (directed) cycles

a dag





not a dag

a,b,c,d,a is a cycle

Topological Sorting

- Topological Sorting: can we list a DAG's vertices as follows?
  - for every edge, its starting vertex is listed before its ending vertex.

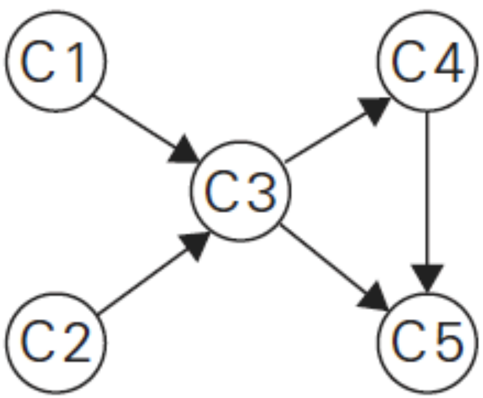
- The topological sorting problem cannot have a solution if a digraph has a directed cycle.
  - Being a dag is necessary for topological sorting.
  - determine a directed graph is a dag or not

- Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

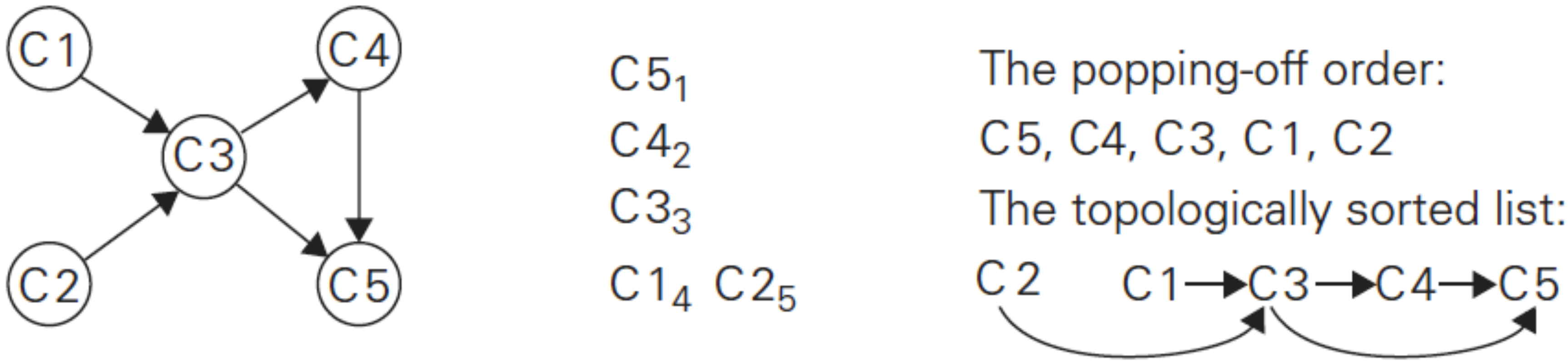
Topological Sorting

- Two algorithms to solve the topological sorting problem
  - DFS-based Algorithm
  - Source Removal Algorithm

- Example: a digraph representing the prerequisite structure of five courses



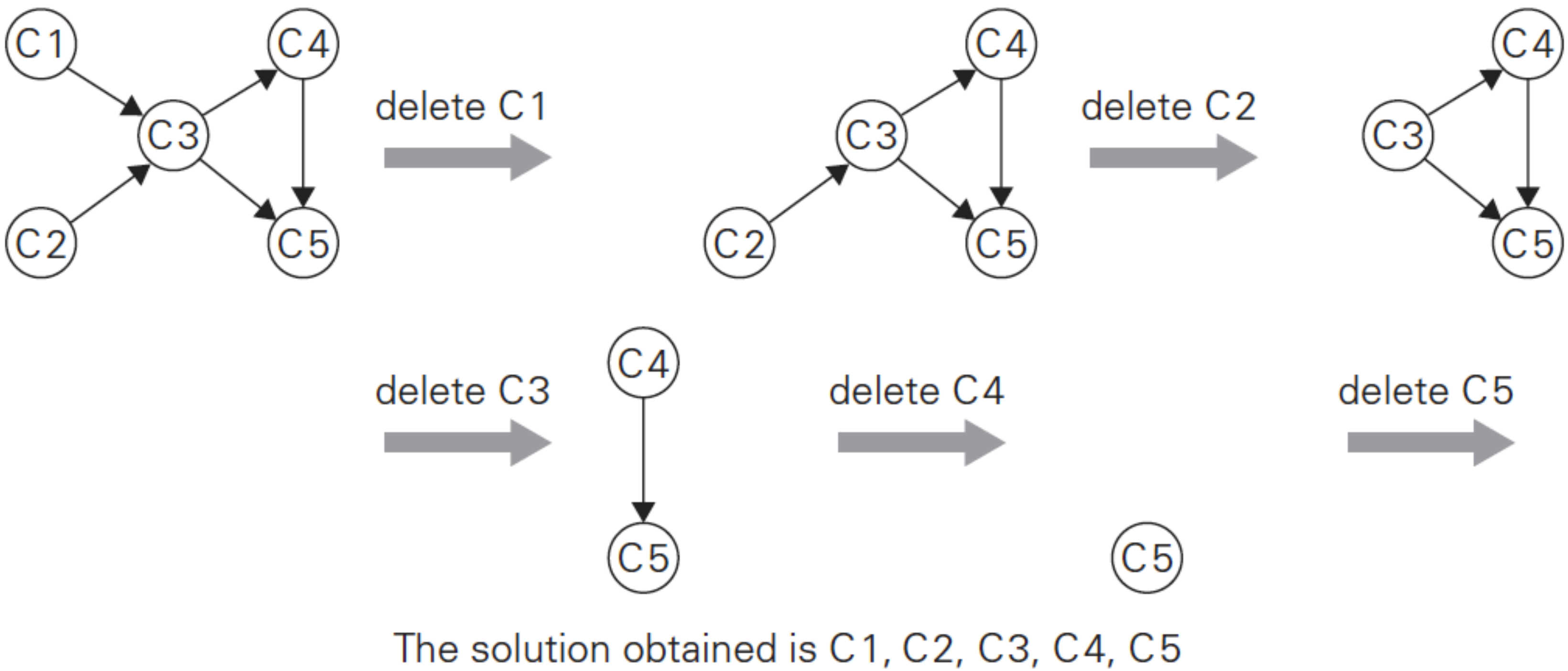
Topological Sorting: DFS - Example



(a) (b) (c)

- When a vertex v is popped off a DFS stack,
  - no vertex u with an edge from u to v can be among the vertices popped off before v.
  - Any such vertex u will be listed after v in the popped-off order list, and before v in the <u>reversed</u> list.

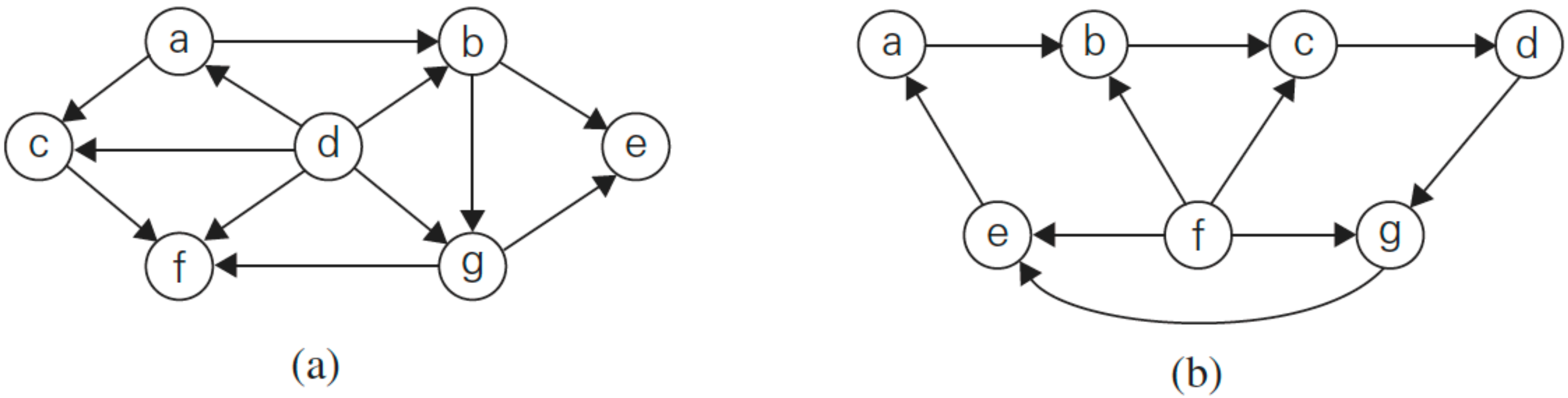
Topological Sorting: Source Removal - Example



- Direct implementation: repeatedly,
  - identify in a remaining digraph a source (vertex with no incoming edges)
  - Delete it along with all the edges outgoing from it.
- The order in which the vertices are deleted yields a solution.

Exercise 4.2

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



4.3 Algorithms for Generating Combinatorial Objects

- We already encountered for exhaustive search

- Most important types:
  - Permutations
  - Combinations
  - Subsets

- A branch of discrete mathematics: combinatorics
- The number of combinatorial objects typically grows exponentially
  (or even faster) as a function of the problem size.

Generating Permutations

- Generating Permutations:
  - Bottom up algorithm
  - Generating all $n!$ permutations of $\{1,\ 2,\ \cdots,\ n\}$,
    first, generate all $(n-1)!$ permutations.
- To satisfy the minimal-change requirement
  - Start with inserting n into $1,\ 2,\ \cdots,\ (n-1)$
  - By moving right to left and then switch direction every time a new permutation of $\{1,\ 2,\ \cdots,\ (n-1)\}$

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

Exercise 4.3

2. Generate all permutations of {1, 2, 3, 4} by

a. the bottom-up minimal-change algorithm.

Johnson-Trotter Algorithm

- Without explicit generating permutations for smaller values of n:
  - Associating a direction with each element k in a permutation.
  - Indicate such a direction by a small arrow written above the element in question.
- Mobile: the element k if its arrow points to a smaller number adjacent to it
  - Example: 3 and 4 are mobile while 2 and 1 are not.

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

Johnson-Trotter Algorithm: Pseudocode

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

- Example: $n = 3$ $\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}}\quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2}\quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}}\quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$
- Efficiency: $\Theta(n!)$
    - One of the most efficient permutation permutation

Exercise 4.3

2. Generate all permutations of {1, 2, 3, 4} by

b. the Johnson-Trotter algorithm.

Lexicographic-order Algorithm

- Lexicographic order: permutations were listed in increasing order
  - For alphabet letters, it would be listed in a dictionary
- 123 132 213 231 312 321.

Lexicographic-order Algorithm: Pseudocode

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

Exercise 4.3

2. Generate all permutations of {1, 2, 3, 4} by

c. the lexicographic-order algorithm.

Generating Subsets: Knapsack problem

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

- Previously, exhaustive-search approach
  - Generate all subsets of a given set of items.

- Let's discuss
  - How can we generate all 2^n subsets of an abstract set $A = \{a_1, a_2, \cdots, a_n\}$

Generating Subsets: Knapsack problem

- How can we generate all 2^n subsets of an abstract set $A = \{a_1, a_2, \cdots, a_n\}$
  - The straight-forward (or bottom up) implementation
  - Let S_{n-1} be the set of all subsets of A,
  - $S_{n-1} = \{A_1, A_2, \cdots, A_m\}, m = 2^{n-1}$
  - $S_n = \{A_1,\ A_2,\ \cdots,\ A_m,\ A_1 \cup a_n,\ A_2 \cup a_n,\ \cdots,\ A_m \cup a_n\}$

n	subsets							
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: Knapsack problem

- How can we generate all 2^n subsets of an abstract set
  $A = \{a_1, a_2, \cdots, a_n\}$
  - The straight-forward (or bottom up) implementation
  - Let S_{n-1} be the set of all subsets of A,
  - $S_{n-1} = \{A_1, A_2, \cdots, A_m\}, m = 2^{n-1}$
  - $S_n = \{A_1,\ A_2,\ \cdots,\ A_m,\ A_1 \cup a_n,\ A_2 \cup a_n,\ \cdots,\ A_m \cup a_n\}$
- All 2^n bit strings $b_1, b_2, \cdots, b_n$ of length n.
  - One-to-one correspondence
    - $b_i = 1$ if a_i belongs to the subset
    - $b_i = 0$ if a_i does not belong to the subset

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

Generating Subsets:Binary reflected Gray code

- Binary reflected Gray code:
  - minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit.

000 001 011 010 110 111 101 100.

base	reflect	prepend	reflect	prepend
0	0	00	00	000
1	1	01	01	001
	1	11	11	011
	0	10	10	010
			10	110
			11	111
			01	101
			00	100

Binary reflected Gray code: Pseudocode

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

Exercise 4.3

5. Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ by each of the two algorithms outlined in this section.