Brute Force

Brute Force

- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- Examples – based directly on definitions

  i. Computing a^n ($a > 0$, n a nonnegative integer)

  ii. Computing $n!$

  iii. Multiplying two matrices

  iv. Searching for a key of a given value in a list

String Matching by Brute Force

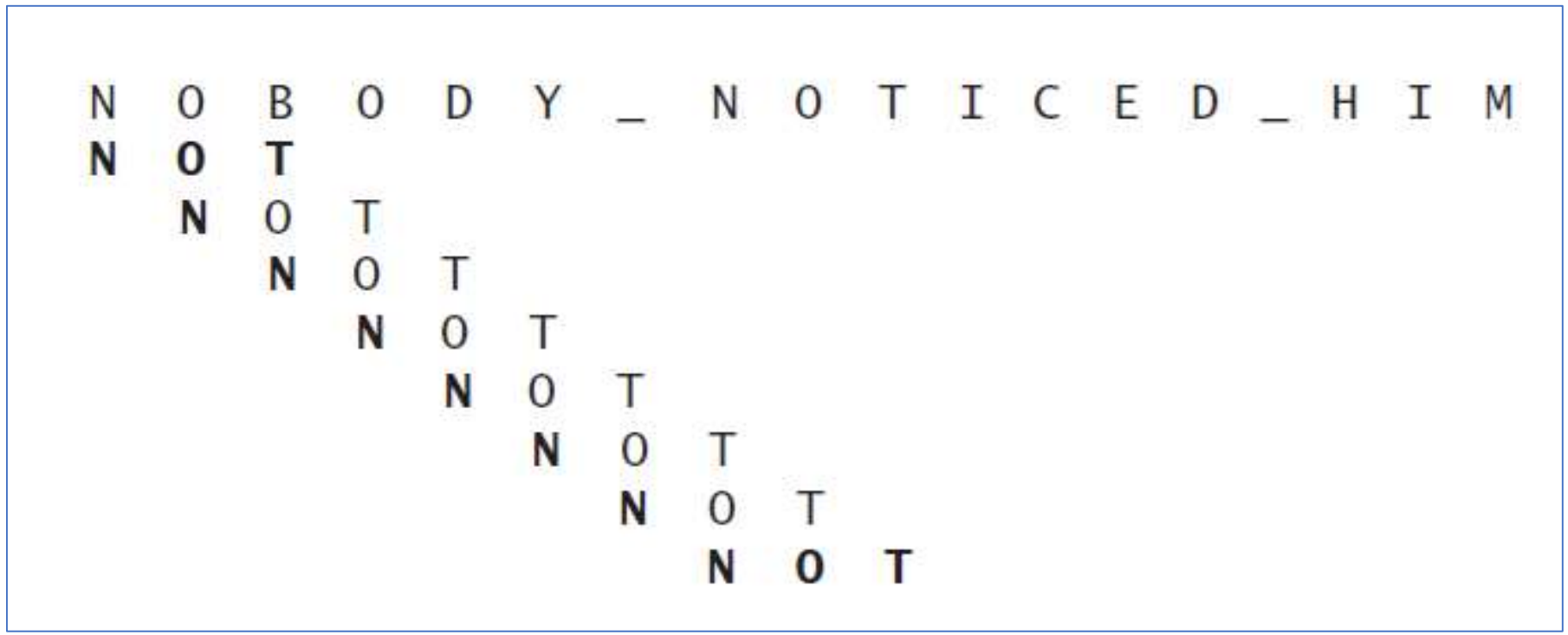
- ***pattern*:** a string of *m* characters to search for
- ***text*:** a (longer) string of *n* characters to search in
- **problem:** find first substring in text that matches pattern

Brute-force: Scan text LR, compare chars, looking for pattern,

Step1 Align pattern at beginning of text

Step2 Moving from left to right, compare each character of pattern to the corresponding character in text until
- all characters are found to match (successful search); or
- a mismatch is detected

Step3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2


N O B O D Y _ N O T I C E D _ H I M
N O T
N O T
N O T
N O T
N O T
N O T
N O T
N O T

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Time Complexity

Worst Case

- The algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.
- In the worst case, the algorithm makes $m(n - m + 1)$ character comparisons, which puts in the $O(nm)$ class.

Average Case

- A typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons
- Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. For searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$.

Exhaustive Search

- ***Exhaustive search*** is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element
- Exhaustive search is a brute-force search algorithm that systematically checks all possible solutions to a problem. It's also known as generate and test.

Approach:

1. Enumerate and evaluate all solutions, and
2. Choose solution that meets some criteria (eg smallest)

Exhaustive Search

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

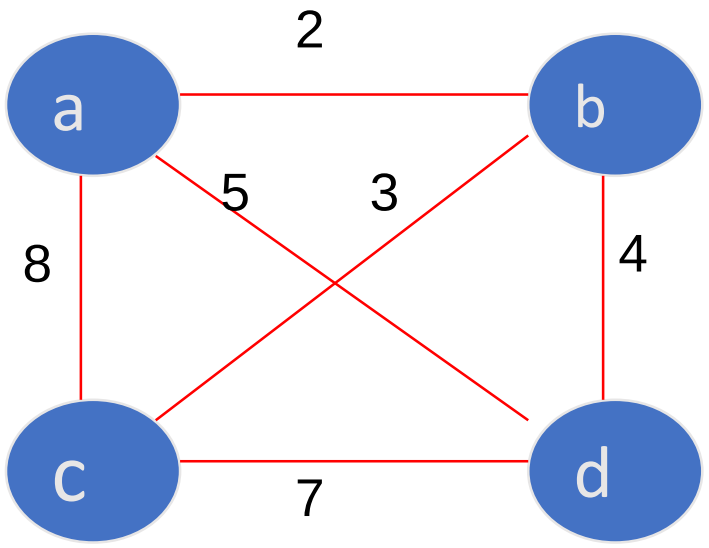
- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- When search ends, announce the solution(s) found

Exhaustive Search –Examples

- Traveling Salesman Problem (TSP)
- Knapsack Problem
- Assignment Problem

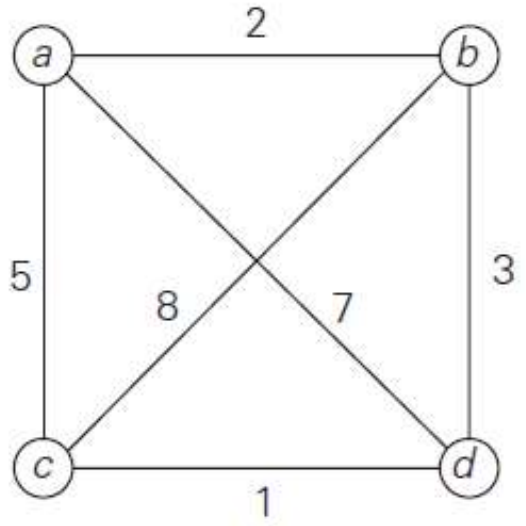
Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- More formally: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

Solution to a small instance of the traveling salesman problem by exhaustive search.

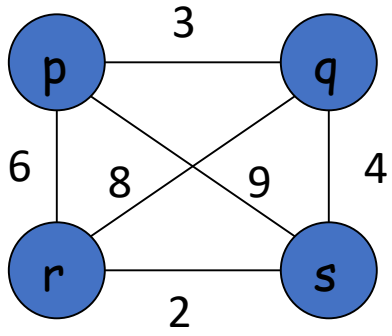


Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Efficiency:

tours
=O(# permutations of b,c,d)
= O(n!)

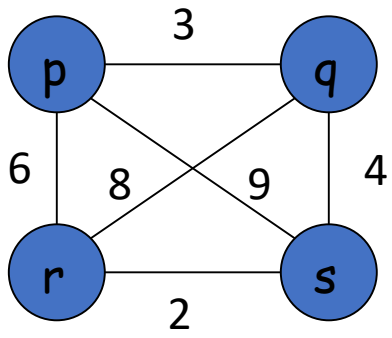
Question



List all tours starting from city p and find the shortest among them

Question

List all tours starting from city p and find the shortest among them



p $\xrightarrow{3}$ q $\xrightarrow{9}$ r $\xrightarrow{2}$ s $\xrightarrow{8}$ p (Cost = 22)

p $\xrightarrow{3}$ q $\xrightarrow{4}$ s $\xrightarrow{2}$ r $\xrightarrow{6}$ p (Cost = 15)

p $\xrightarrow{6}$ r $\xrightarrow{9}$ q $\xrightarrow{4}$ s $\xrightarrow{8}$ p (Cost = 27)

p $\xrightarrow{6}$ r $\xrightarrow{2}$ s $\xrightarrow{4}$ q $\xrightarrow{3}$ p (Cost = 15)

p $\xrightarrow{8}$ s $\xrightarrow{4}$ q $\xrightarrow{9}$ r $\xrightarrow{6}$ p (Cost = 27)

p $\xrightarrow{8}$ s $\xrightarrow{2}$ r $\xrightarrow{9}$ q $\xrightarrow{3}$ p (Cost = 22)

Example 2: Knapsack Problem

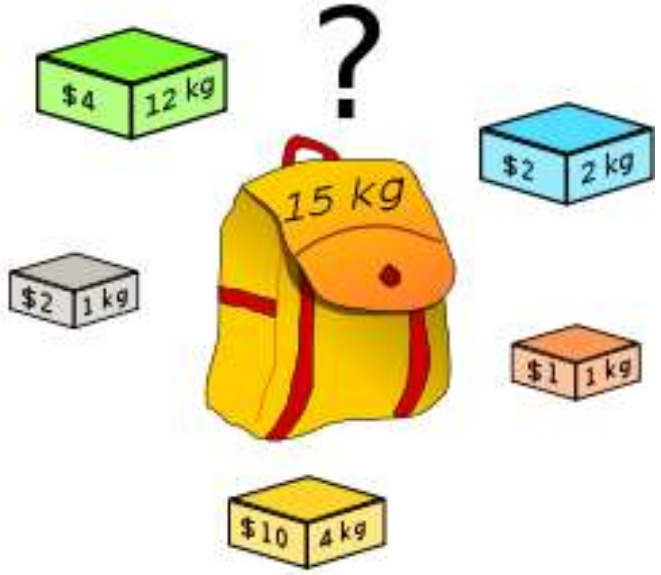
Given *n* items:

- weights: w_1 w_2 … w_n
- values: v_1 v_2 … v_n
- a knapsack of capacity *W*

Find most valuable subset of the items that fit into the knapsack

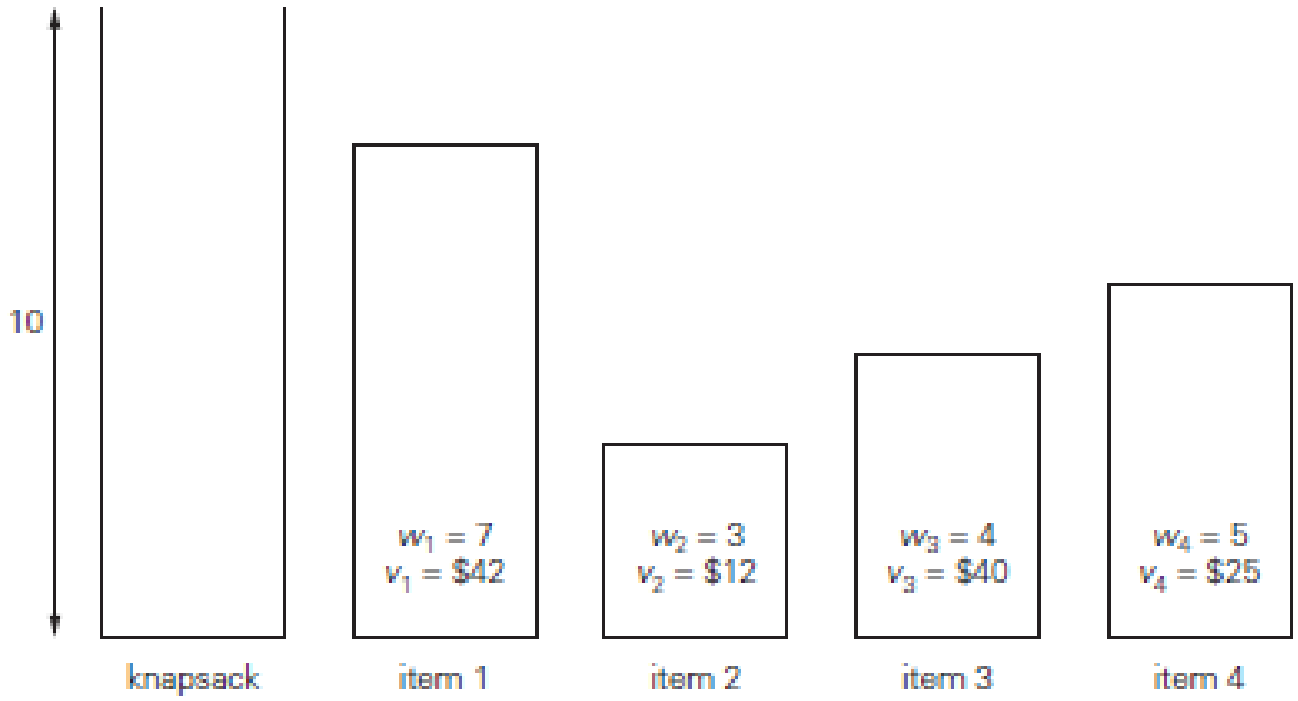
Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10



Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency: how many subsets?



Subset	Total weight	Total value
∅	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	**9**	**\$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Question: Knapsack Problem

- Let W=40. Let the weight of three objects be 20, 25, 10 and value of corresponding objects be 30, 40 and 35. Find the optimal solution.

Sl. Number	Objects selected	Total weight of objects selected	Feasible or not	Profit earned
1.	None i.e., { }	0	Feasible	0
2.	1 i.e., {1}	20	Feasible	30
3.	2 i.e., {2}	25	Feasible	40
4.	3 i.e., {3}	10	Feasible	35
5.	1,2 i.e., {1, 2}	20 + 25 = 45	Not feasible	
6.	1,3 i.e., {1,3}	20 + 10 = 30	Feasible	65
7.	2,3 i.e., {2, 3}	25 + 10 = 35	Feasible	75
8.	1,2,3 i.e., {1,2,3}	20 + 25 + 10 = 55	Not feasible	

Example 3: The Assignment Problem

There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

Algorithmic Plan:

Generate all legitimate assignments

Compute costs

Select cheapest

$$C = \begin{matrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{matrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
…	…

$$C = \begin{matrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{matrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
…	…

(For this instance, the optimal assignment can be easily found by exploiting the specific features of the numbers given. It is: (2, 1, 3, 4)

Example: Assignment Problem

	Job 1	Job 2	Job 3	Job 4
Person A	10	3	8	9
Person B	7	5	4	8
Person C	6	9	2	9
Person D	8	7	10	5

<1,2,3,4> Cost = 10 + 5 + 2 + 5 = 22
<1,2,4,3> Cost = 10 + 5 + 9 + 10 = 34
<1,3,2,4> Cost = 10 + 4 + 9 + 5 = 28
<1,3,4,2> Cost = 10 + 4 + 9 + 7 = 30
<1,4,2,3> Cost = 10 + 8 + 9 + 10 = 37
<1,4,3,2> Cost = 10 + 8 + 2 + 7 = 27

<2,1,3,4> Cost = 3 + 7 + 2 + 5 = 17
<2,1,4,3> Cost = 3 + 7 + 9 + 10 = 29
<2,3,1,4> Cost = 3 + 5 + 6 + 5 = 19
<2,3,4,1> Cost = 3 + 5 + 9 + 8 = 25
<2,4,1,3> Cost = 3 + 8 + 6 + 10 = 27
<2,4,3,1> Cost = 3 + 8 + 2 + 8 = 21

<3,1,2,4> Cost = 8 + 7 + 9 + 5 = 29
<3,1,4,2> Cost = 8 + 7 + 9 + 7 = 31
<3,2,1,4> Cost = 8 + 5 + 6 + 5 = 24
<3,2,4,1> Cost = 8 + 5 + 9 + 8 = 30
<3,4,1,2> Cost = 8 + 8 + 6 + 7 = 29
<3,4,2,1> Cost = 8 + 8 + 9 + 8 = 33

<4,1,2,3> Cost = 9 + 7 + 9 + 10 = 35
<4,1,3,2> Cost = 9 + 7 + 2 + 7 = 25
<4,2,1,3> Cost = 9 + 5 + 6 + 10 = 30
<4,2,3,1> Cost = 9 + 5 + 2 + 8 = 24
<4,3,1,2> Cost = 9 + 4 + 6 + 7 = 26
<4,3,2,1> Cost = 9 + 4 + 9 + 8 = 30

Decrease and Conquer

Decrease and Conquer - Introduction

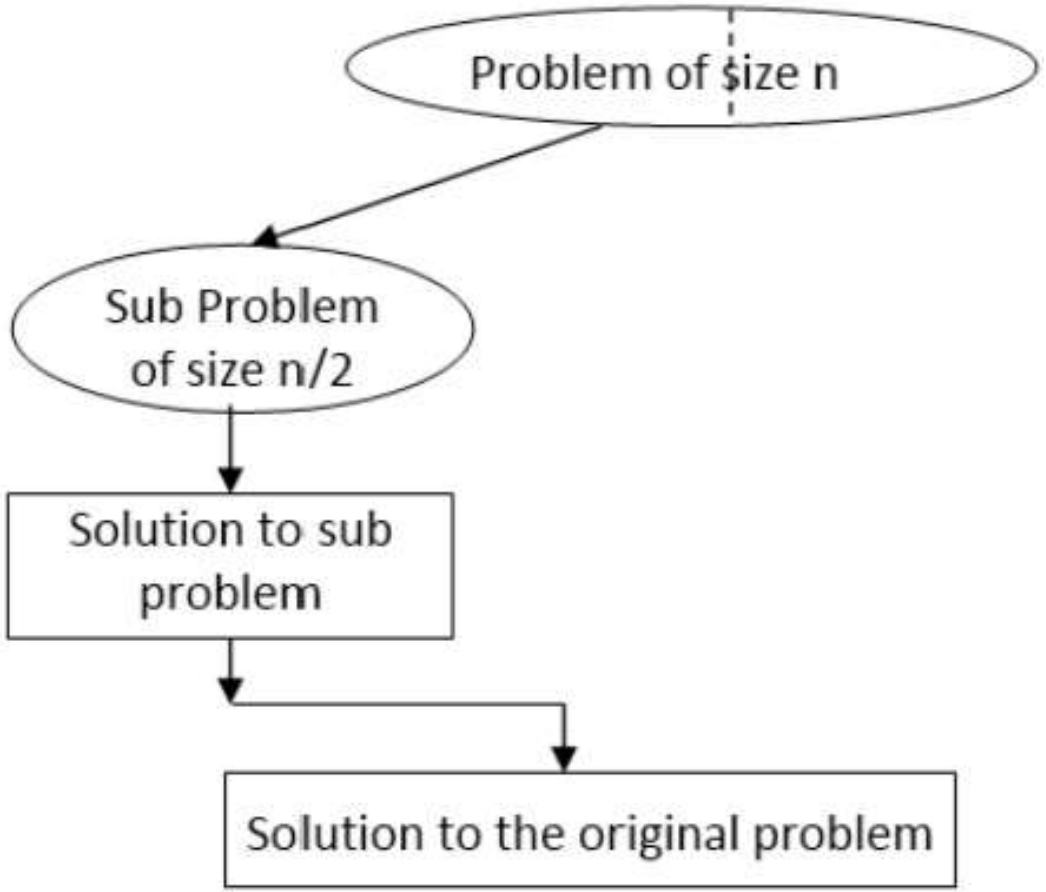
- The Decrease and Conquer technique is based on Problem reduction Strategy
- It is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- The exploitation can be either top-down (recursive) or bottom-up (non-recursive)

Decrease and Conquer Variation

- There are three major variations of decrease-and-conquer
  - Decrease-by-a-constant
  - Decrease-by-a-constant-factor
  - Variable-size-decrease

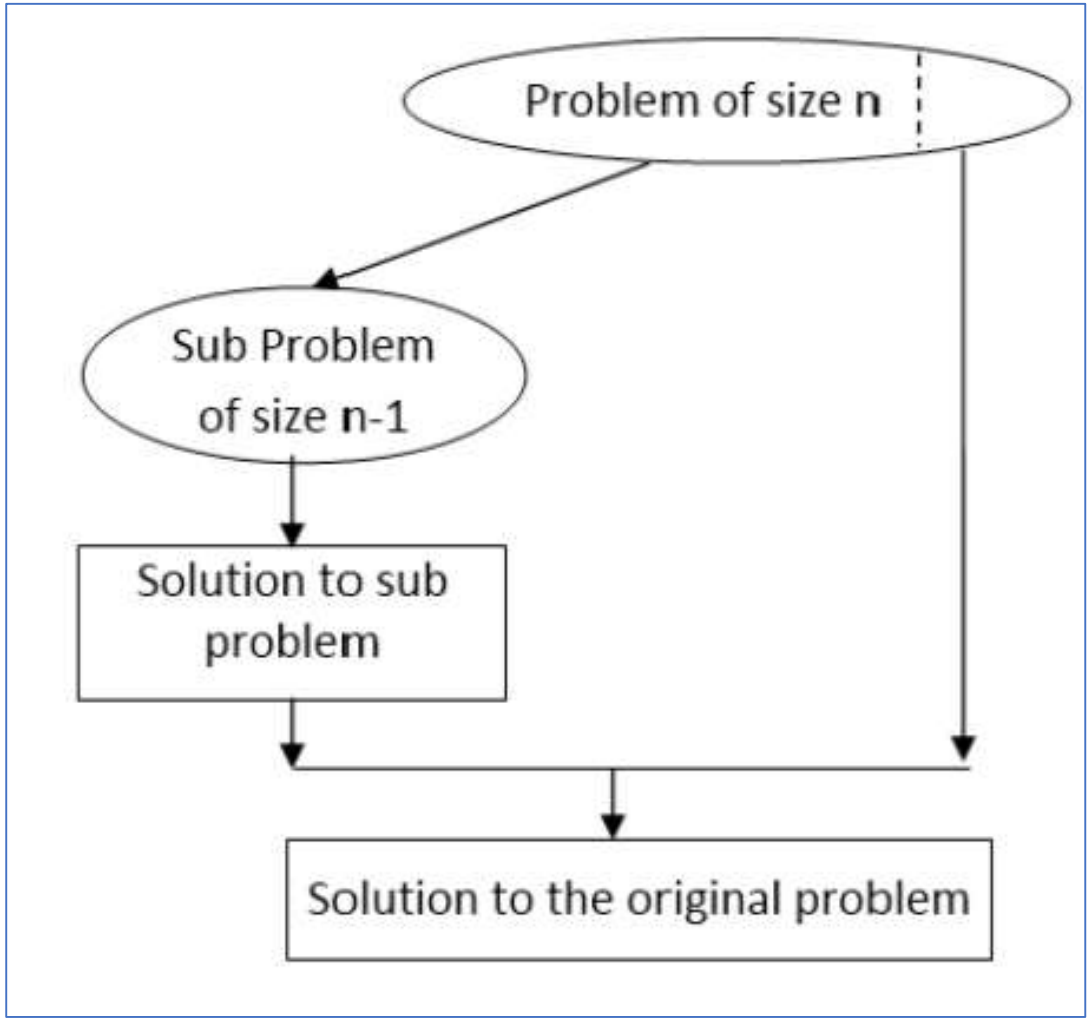
Decrease by Constant Factor

- Technique suggest that reducing a problem's instance by the same constant factor on each iteration of the algorithm
- In most cases the constant factor is equal to two



Decrease by Constant

- The size of the instance is reduced by the same constant value on each iteration of the algorithm
- Typically this constant is equal to one



Decrease by Constant

- Example: Exponentiation problem of computing a^n for positive integer exponent
- The relationship between solution to an instance of problem of size n and an instance of problem of size n-1 is obtained by the formula

$$a^n = a^{n-1} . a$$

The function f(n) = an can be computed using recursive definition as

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

Decrease by Constant Factor

- Consider the exponential problem
- The function f(n)= a^n can be computed as

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

Decrease by Constant Factor - Binary Search

$$\underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad \begin{matrix} K \\ \updownarrow \\ A[m] \end{matrix} \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}.$$

Variable Size Decrease

- The size reduction pattern varies from one iteration of the algorithm to the another
- Example
  - Algorithm for Computing GCD

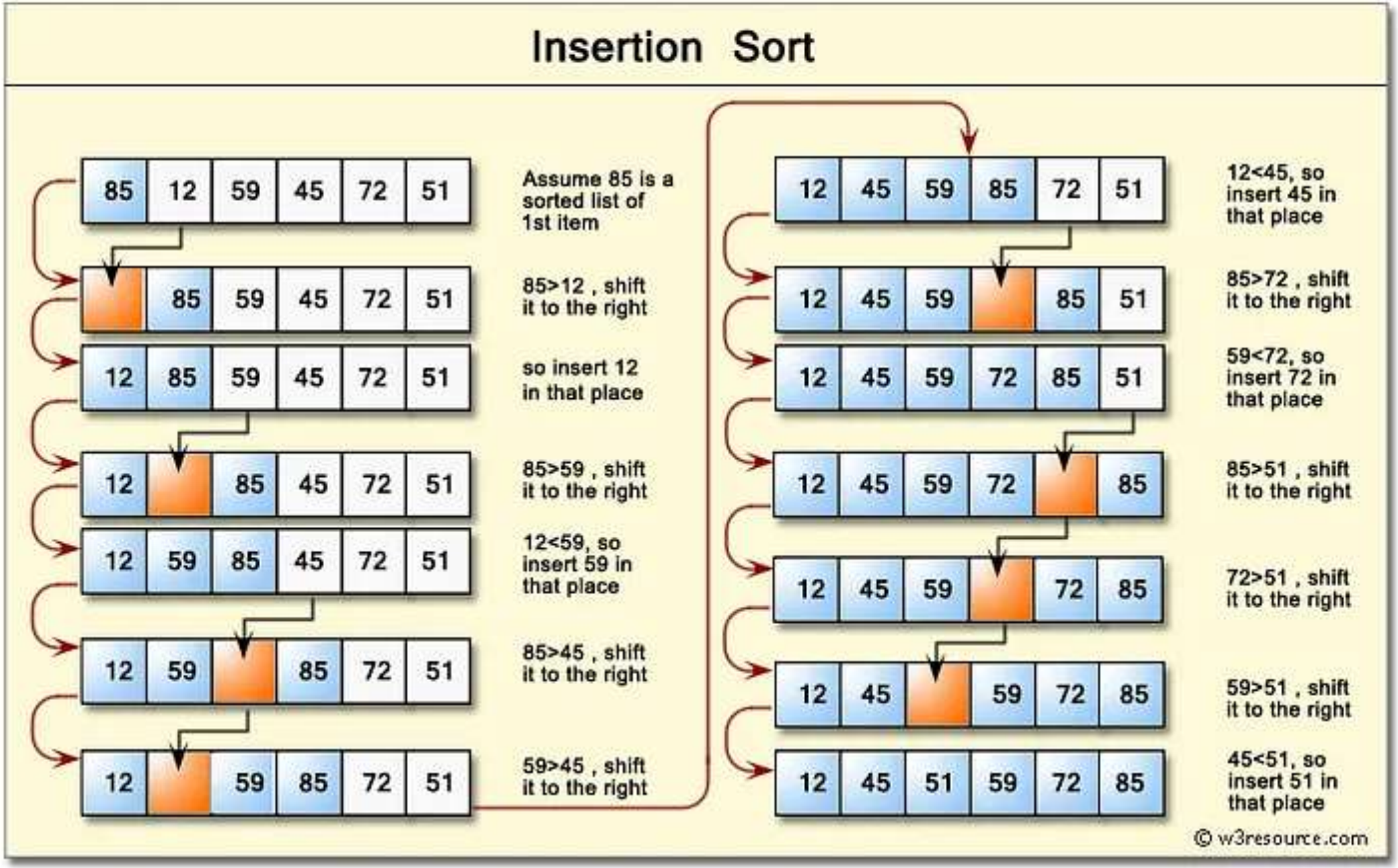
$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Insertion Sort

- Insertion sort is based on decrease and conquer design approach.
- Its essential approach is to take an array A[0..n-1] and reduces its instance by a factor of 1, Then the instance A[0..n-1] is reduced to A[0..n-2].
- This process is repeated till the problem is reduced to a small problem enough to get solved.

Insertion Sort is one of the elementary sorting algorithms used to sort a list of elements in ascending order. It works by building a sorted section from left to right, one element at a time, and inserting each new element into its correct position within the sorted section.


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

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
```

Worst Case Time Complexity

- The number of key comparisons in this algorithm obviously depends on the nature of the input.
- In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$.
- Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$.
- Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), $\ldots, A[n-2] > A[n-1]$ (for $i = n - 1$).
- In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

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

Best case Complexity

- In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop.
- It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots, n-1$, i.e., if the input array is already sorted in nondecreasing order.
- The best case of an algorithm happens when the problem is already solved

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

Average Case

- A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order
- It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays

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

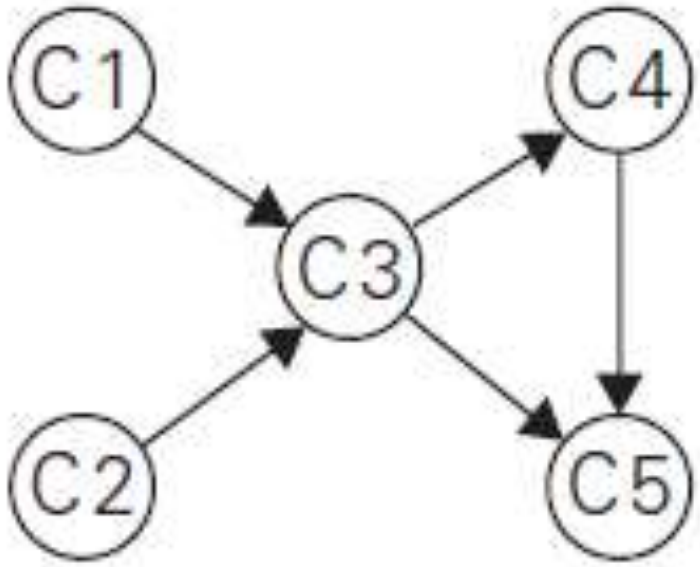
Topological sorting

- Topological sort is a technique used in graph theory to linear ordering of the vertices of a directed acyclic graph (DAG).

- It ensures that for every directed edge from vertex A to vertex B, vertex A comes before vertex B in the ordering.

- This is useful in scheduling problems, where tasks depend on the completion of other tasks.
- **Note: Topological Sorting for a graph is not possible if the graph is not a DAG.**

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 course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

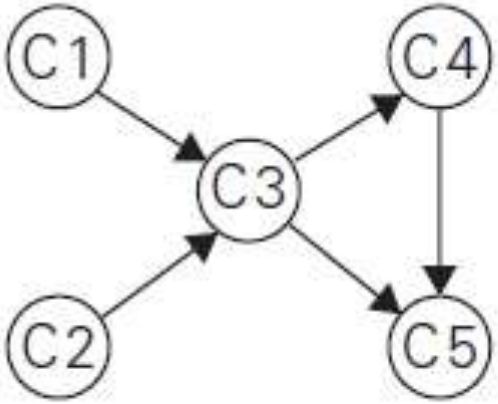


Digraph representing the prerequisite structure of five courses

DFS -Depth-First Search

- Perform a DFS traversal and note the order in which **vertices become dead-ends** (i.e., popped off the traversal stack).
- Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal.
- If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Topological sort = Reversal of the order in which the vertices become dead ends in the DFS algorithm.



(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:
C2 C1→C3→C4→C5

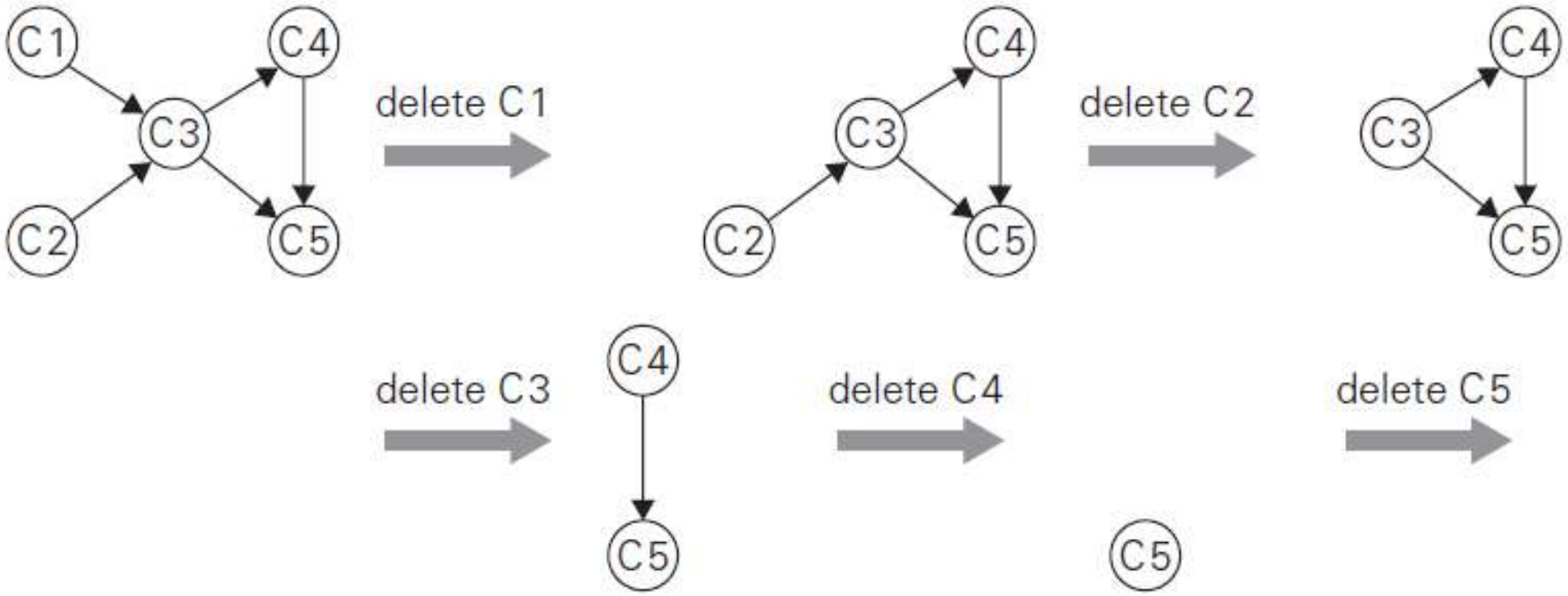
(c)

(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.

Source-Removal Algorithm

- The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique
- Repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. section's exercises.)
- The order in which the vertices are deleted yields a solution to the topological sorting problem.

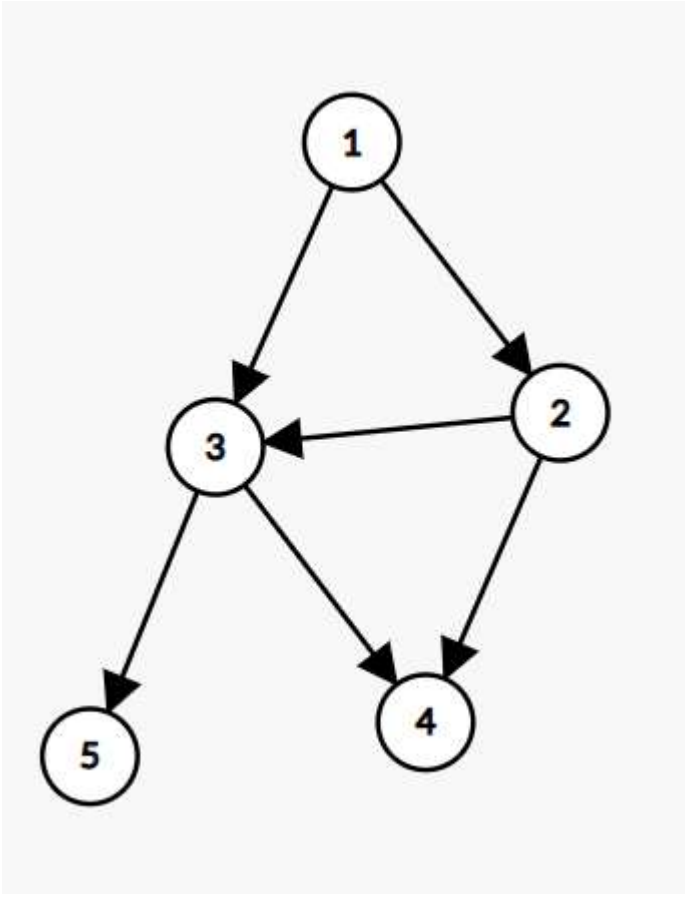
Source-Removal Algorithm



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

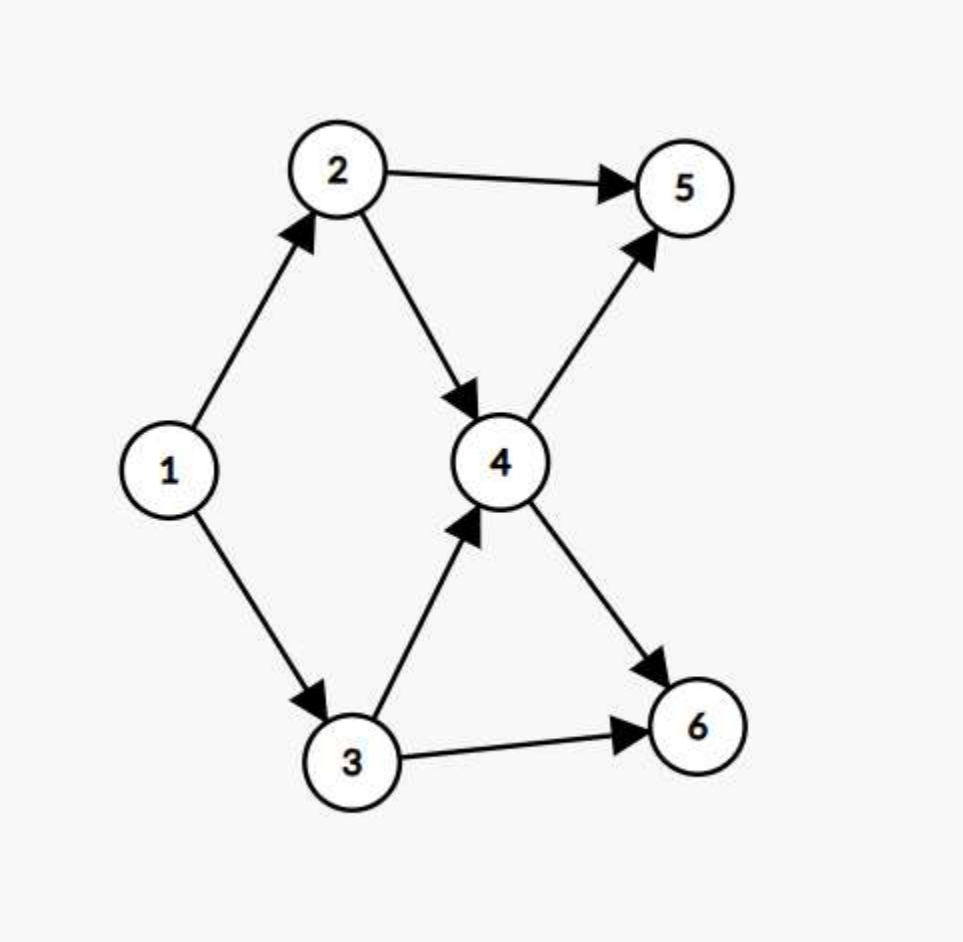
Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Topological Sorting -DFS



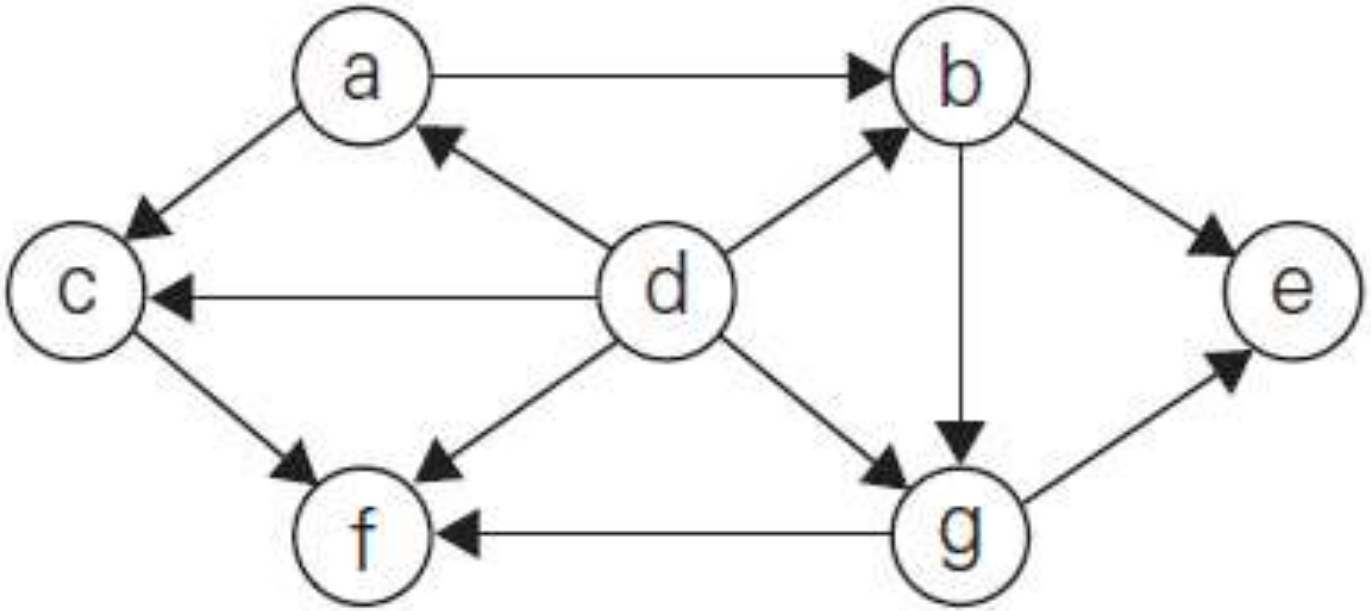
One of the solution is : 1 2 3 5 4

Topological Sorting -DFS



1, 2, 3, 4, 5, 6

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



d,a,c,b,g,f,e

Prove that Topological sorting problem has a solution if and only if it is a DAG

Generating Combinatorial Objects

- All the permutation of a set of objects
- The most important types of combinatorial objects are permutations, combinations and subsets of a given set.

Generating Permutations

- A permutation of a set is an ordering of all its elements.
- For example, for the set {a, b, c, T} we can define two different permutations (but there are more, of course} a, c, T, b and T, c, b, a.
- If the elements of a set are indexed, then a permutation of indices uniquely determines a permutation of the set's elements. For example, index the elements of the set {a, b, c, T} with numbers 1, 2, 3, 4.
- Then its two sample permutations above correspond to the two permutations of indices, 1, 3, 4, 2 and 4, 3, 2, 1.
- It follows that as soon as we know how to get permutations of a set of indices {1 2 3 ... N}, for a given N, we know how to permute any set with N elements.

- There are many ways to list all permutations of the given length N.
- A most natural approach builds permutations of growing length successively starting with the shortest ones.
- A 1-element set, say {1}, has only one permutation: 1.
- Moving to the set {1, 2} of two elements, we see that the additional element 2 can be appended to the permutation of {1} in two ways: on the left and on the right. Thus we get two permutations:

1 2 and 2 1 ----------(1)

Two elements in each of the permutations in (1) define 3 positions for the 3rd element 3:

3 1 2, 1 3 2, 1 2 3 and 3 2 1, 2 3 1, 2 1 3

- 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

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

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

1.Initialize the first permutation with <1 <2 ... < n
2.Find the largest mobile integer k
3.Swap k and the adjacent integer it is looking at
4.Reverse the direction of all integers larger than k
5.Repeat step 1 until there is no mobile integer left in the sequence

1,2,3, it is the first permutation

$1^{\leftarrow}\ 2^{\leftarrow}\ \mathbf{3}^{\leftarrow}$, Note 2 and 3 are mobile, 3 is the largest, swap it with 2, write it down. Any integers larger than 3?

$1^{\leftarrow}\ \mathbf{3}^{\leftarrow}\ 2^{\leftarrow}$, Only 3 is mobile, swap it with 1, and write

$3^{\leftarrow}\ 1^{\leftarrow}\ \mathbf{2}^{\leftarrow}$, Now 2 is mobile and largest, swap with 1, don't for get to change the direction of 3 because it is larger

$\mathbf{3}^{\rightarrow}\ 2^{\leftarrow}\ 1^{\leftarrow}$, Now 3 is mobile again, swap with 2

$2^{\leftarrow}\ \mathbf{3}^{\rightarrow}\ 1^{\leftarrow}$, Again 3 is mobile, swap with one

$2^{\leftarrow}\ 1^{\leftarrow}\ 3^{\rightarrow}$, None are mobile, all done!

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

Russian Peasant Multiplication

- A nonorthodox algorithm for multiplying two positive integers called ***multiplication `a la russe*** or the ***Russian peasant method***
- Let *n* and *m* be positive integers whose product we want to compute
- let us measure the instance size by the value of *n*. Now, if *n* is even, an instance of half the size has to deal with *n*/2.
- The formula relating the solution to the problem's larger instance to the solution to the smaller one:

The formula relating the solution to the problem's larger instance to the solution to the smaller one:

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

If n is odd, we need only a slight adjustment of this formula:

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

Computing $50 \cdot 65$ by the Russian peasant method.

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

Computing a Median and the Selection Problem

Finds the k-th smallest element in an unordered list

```
List =[7,10,4,3,20,15]
print (quick_select(3,list)

Output
7
```

Lomuto Partition

- Take a random element, this element will be the pivot
- Rearrange the element in the list in each iteration
  - All the elements smaller than Pivot is on the left of pivot
  - All the elements larger than Pivot is on the right of pivot

Input : 4 1 10 8 7 12 9 2 15

Partition: 2 1 4 8 7 12 9 10 15

ALGORITHM *Quickselect*$(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k - 1 - s)$

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//       indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```

Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15. Here, $k = 9/2 = 5$ and our task is to find the 5th smallest element in the array.

0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*i*	
4	1	2	8	7	12	9	10	15
		s						*i*
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

0	1	2	3	4	5	6	7	8
			s	*i*				
			8	7	12	9	10	15
				s	*i*			
			8	7	12	9	10	15
				s				*i*
			8	7	12	9	10	15
			7	**8**	12	9	10	15