Module-2

Divide and Conquer

INTRODUCTION

Divide-and-conquer algorithms work according to the following general plan:
1. Divide: A problem is divided into several subproblems of the same type, ideally of about equal size.
2. Conquer: The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. Combine: If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique is shown in Figure

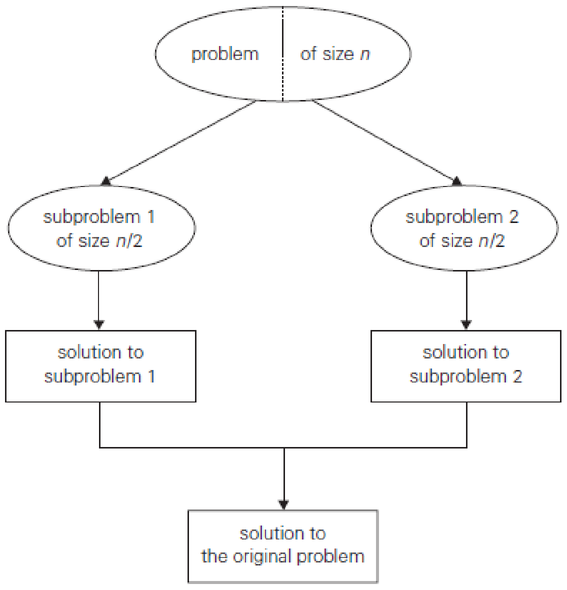

problem
of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

FIGURE 5.1 Divide-and-conquer technique (typical case).

In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size n/2. More generally, an instance of size n can be divided into 'b' instances of size n/b, with 'a' of them needing to be solved.

$$T(n) = aT(n/b) + f(n)$$

where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. Recurrence relation is called the **general divide-and-conquer recurrence.**

Some of the algorithms that make use of divide and conquer approach are Binary search, Quick Sort, Merge Sort, Defective chess board problem etc.,

Master theorem

Theorem: If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation

$$T(n) = aT(n/b) + f(n),$$

then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Example:

Let $T(n) = 2T(n/2) + 1$, solve using master theorem.

Solution:

Here: $a = 2$
$b = 2$
$f(n) = \Theta(1)$

$d = 0$

Therefore:
$a > b^d$ i.e., $2 > 2^0$
Case 3 of master theorem holds good. Therefore:
$T(n) \in \Theta(n^{\log_b a})$
$\in \Theta(n^{\log_2 2})$
$\in \Theta(n)$

Mergesort : Mergesort sorts a given array A[0...n − 1] by dividing it into two halves $A[0... \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor ...n - 1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

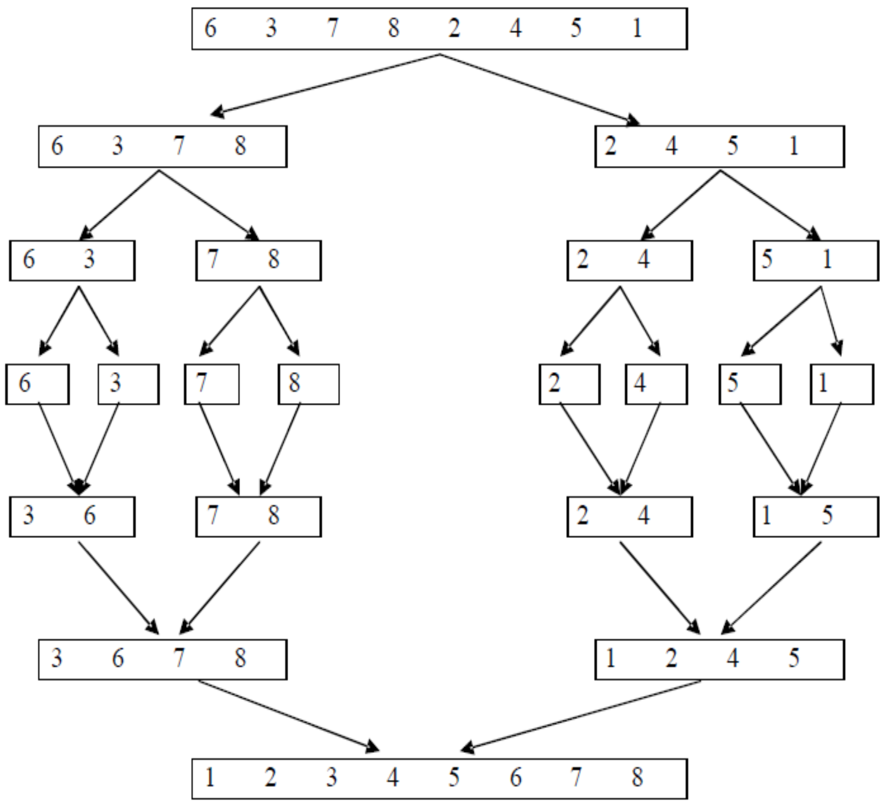
```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)    //see below
```

```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    i ← 0; j ← 0; k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j]
            A[k] ← B[i]; i ← i + 1
        else A[k] ← C[j]; j ← j + 1
        k ← k + 1
    if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
    else copy B[i..p − 1] to A[k..p + q − 1]
```

The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Example:

Apply merge sort for the following list of elements: 6, 3, 7, 8, 2, 4, 5, 1



Analysis of Merge Sort

- Running time T(n) of Merge Sort:
- Divide: computing the middle takes Θ(1)
- Conquer: solving 2 subproblems takes 2T(n/2)
- Combine: merging n elements takes Θ(n)
- Total:

$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \text{ if } n > 1$$

Using Master Theorem , a=2,b=2 d=1, Hence, **T(n) = Θ(n log n)**

Quick Sort:

Quicksort is based on the divide-and conquer approach.

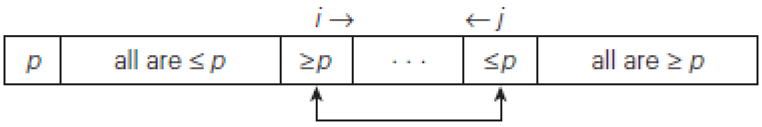
- A partition is used to divide the problem that is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

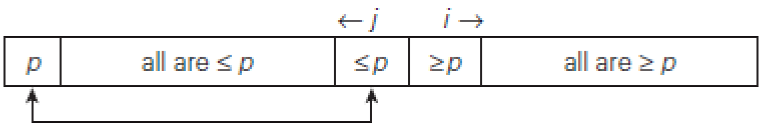
- The strategy used for selecting the pivot element is : the subarray's first element: p = A[l]
- Scan the subarray from both ends, comparing the subarray's elements to the pivot.
- The **left-to-right scan**, denoted by index i, starts with the second element. This scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.

- The **right-to-left scan**, denoted by index j, starts with the last element of the subarray. This scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
-

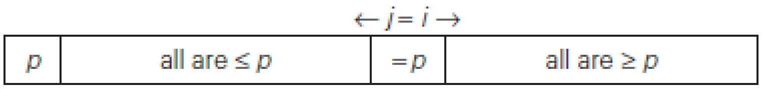
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



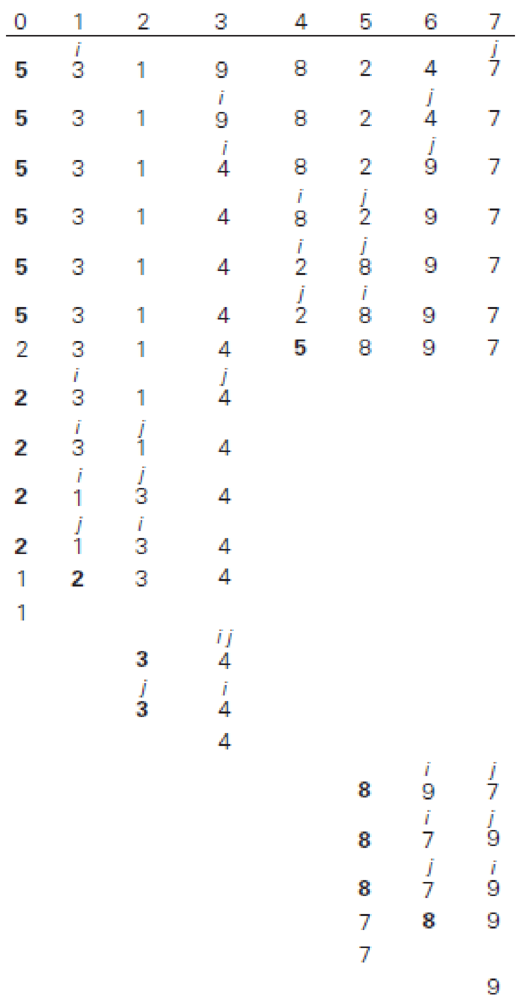
If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:

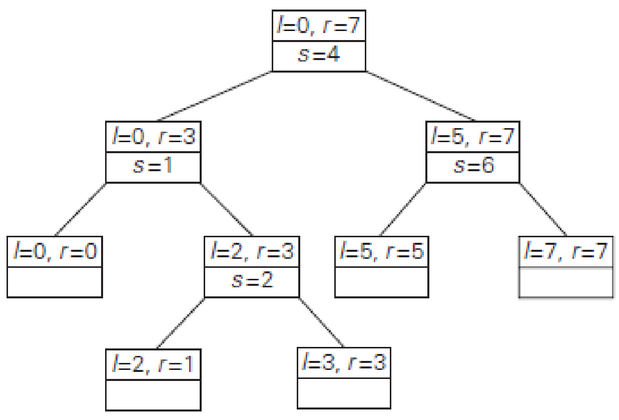


Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.





(b)

Fig (a) shows the working of Quicksort. Fig(b) shows the Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.

Pseudo code

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

```
ALGORITHM HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: Partition of A[l..r], with the split position returned as
    //        this function's value
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])    //undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
```

Best case input: If all the partitions happen in the middle of corresponding subarrays

Worst case input: For increasing arrays, i.e for inputs that are already solved.

Best case efficiency:
Number of key comparisons is n + 1 if the scanning indices cross over
and is n if they coincide
The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

Worst case efficiency:
In the worst case, all the partitions will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned.
This will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved
If we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average case efficiency:
Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after n+1comparisons are made to achieve the partition.
After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Strassen's Matrix Multiplication

Strassen in 1969 which gives an overview that how we can find the multiplication of two 2*2 dimension matrix by the brute-force algorithm. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. Let A and B be two n × n matrices where n is a power of 2.We can divide A,B, and their product C into four n/2 × n/2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

Let us evaluate the asymptotic efficiency of this algorithm. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots \\ &= 7^iM(2^{k-i}) \cdots = 7^kM(2^{k-k}) = 7^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

Decrease-and-Conquer

This algorithm design technique is based on exploiting a relationship between a solution to a given instance of the problem in question and its smaller instance.

Once such a relationship is found, it can be exploited either top down (usually recursively) or bottom up.

3 Types of Decrease and Conquer

1. Decrease by a constant (usually by 1):
   - insertion sort
   - topological sorting
   - algorithms for generating permutations, subsets
2. Decrease by a constant factor (usually by half)
   - binary search and bisection method
   - exponentiation by squaring
3. Variable-size decrease
   - Euclid's algorithm
   - selection by partition

1. In the **decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.

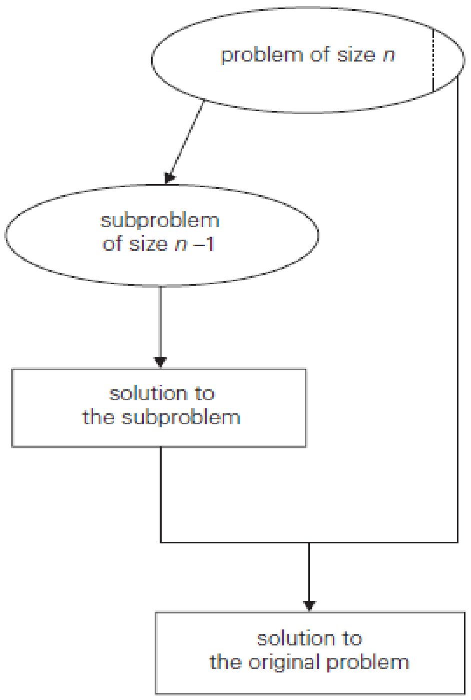


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

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

Or "bottom up" by multiplying 1 by a n times.

2. The **decrease-by-a-constant-factor** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.

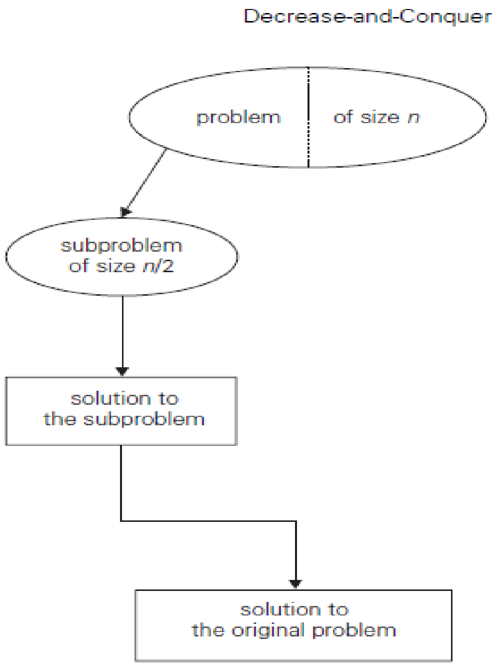


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

If the instance of size n is to compute an, the instance of half its size is to compute $a^{n/2}$, with the relationship between the two: $a^n = (a^{n/2})^2$.

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

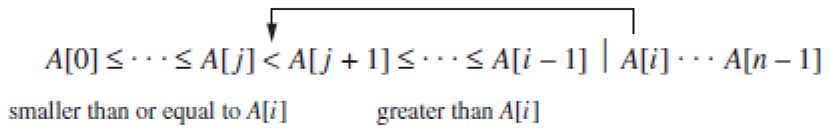
3. In the variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation.

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Insertion Sort

- an application of the decrease-by-one technique to sorting an array A[0..n − 1]
- Algorithm is implemented bottom up, i.e., iteratively.

- As shown in Figure, starting with A[1]and ending with A[n − 1], A[i] is inserted in its appropriate place among the first i elements of the array that have been already sorted


$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \mid A[i] \cdots A[n-1]$
smaller than or equal to $A[i]$
greater than $A[i]$

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 array from the remaining elements; the element being inserted is in bold.

Here is pseudocode of this 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 $j \ge 0$ and $A[j] > v$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$
```

Analysis for Insertion Sort:

The basic operation of the algorithm is the key comparison A[j]>v.

Worst case input and efficiency:
The worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

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

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort

Best case input and efficiency:
In the best case, the comparison A[j]> v is executed only once on every iteration of the outer loop. This happens for sorted arrays. Thus, for sorted arrays, the number of key comparisons is

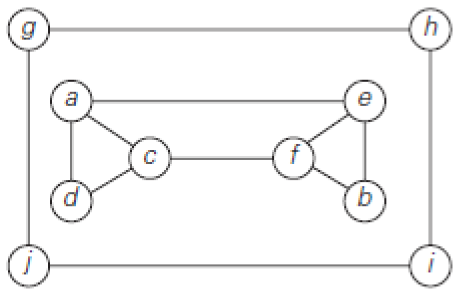
$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Average-case efficiency
on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

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

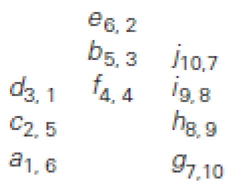
Depth-First Search

Depth-first search starts a graph's traversal at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a dead end—a vertex with no adjacent unvisited vertices-is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.
It is convenient to use a stack to trace the operation of depth-first search.We push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts), and we pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).
A depth-first search traversal is accompanied by 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.



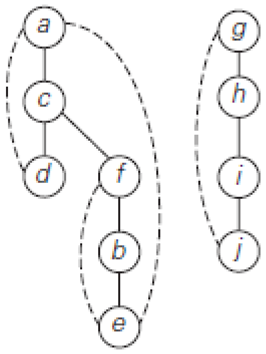
(a)

Graph



(b)

Traversal Stack



(c)

DFS Forest

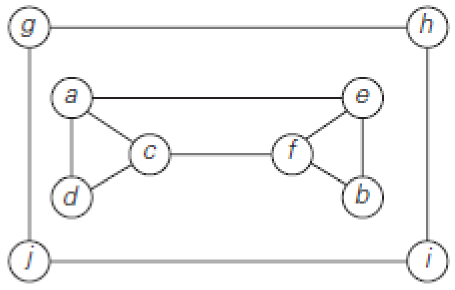
```
ALGORITHM DFS(G)
    //Implements a depth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph G with its vertices marked with consecutive integers
    //        in the order they are first encountered by the DFS traversal
    mark each vertex 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)

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)
```

Applications of DFS

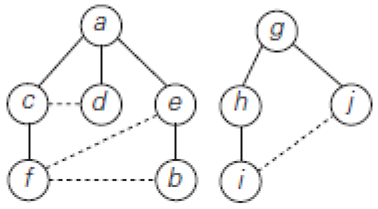
1. To check whether a given graph is connected or not
2. To check whether a given graph is acyclic or not
3. To find the spanning tree
4. Topological sorting
5. To find articulation points in a graph

Breadth-First Search

BFS proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
It is convenient to use a queue to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.
A BFS traversal is by accompanied by 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 is encountered, the edge is noted as a **cross edge**.



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



(a) Graph

(b) Traversal Queue

(c) BFS Forest

```
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 are 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)

    bfs(v)
    //visits all the unvisited vertices connected to vertex v
    //by a path and numbers them 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
```

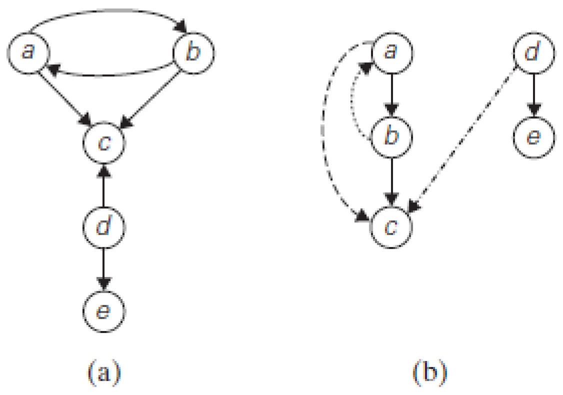
Applications of BFS

1. To check whether a given graph is connected or not
2. To check whether a given graph is acyclic or not
3. To find the spanning tree
4. To find minimum-edge paths

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one 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 adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Topological Sorting

A **directed graph,** or **digraph** for short, is a graph with directions specified for all its edges.The adjacency matrix and adjacency lists are still two principal means of representing a digraph.



a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

A **directed cycle** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, a, b, a is a directed cycle in the digraph in Figure.
Conversely, if a DFS forest of a digraph has no back edges, the digraph is a dag, an acronym for directed acyclic graph.
Topological sorting is to list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.
For topological sorting to be possible, a digraph in question must be a dag.

Topological sorting methods:
1) DFS method
2) Source Removal method

1) DFS method

- an application of 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, andtopological sorting of its vertices is impossible

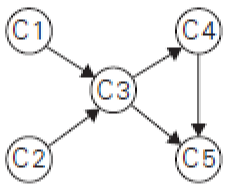


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

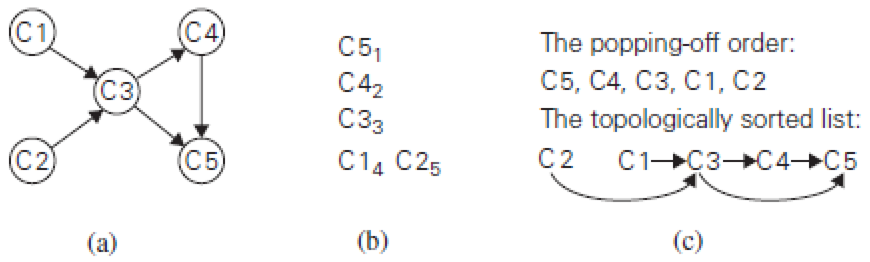


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

2) Source Removal Method

- Based on a direct implementation of the decrease-(byone)-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 none, stop because the problem cannot be solved
- The order in which the vertices are deleted yields a solution to the topological sorting problem.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

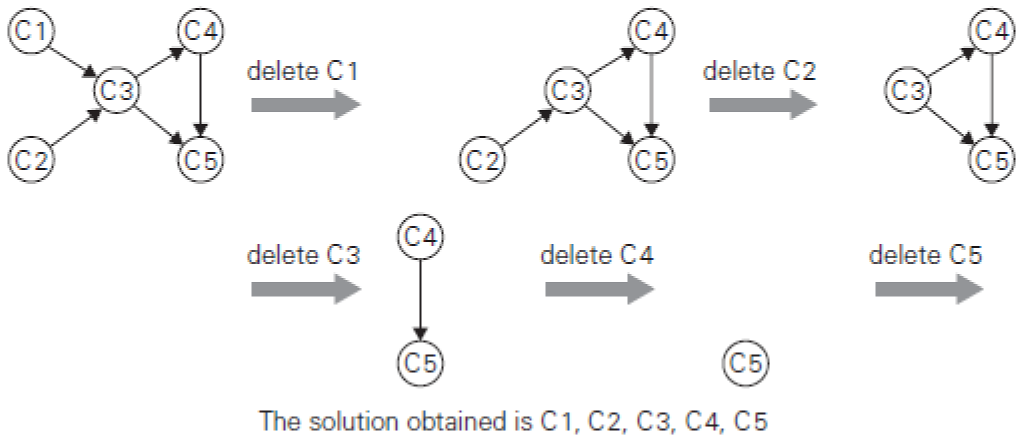


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Algorithms for Generating Combinatorial Objects

There are three different types of combinatorial objects
1) Permutations
2) Combinations
3) Subsets of a given set

1) Generating Permutations : The permutations can be generated using various methods such as:
- Bottom-up minimal change algorithm
- Johnson Trotter algorithm
- Lexicographic ordering

Bottom-up minimal change algorithm: The decrease-by-one technique suggest that for the problem of generating all n! permutations of {1, . . . , n}, the smaller-by-one problem to generate all (n − 1)! Permutations is to be solved. Assuming that the smaller problem is solved, we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of n − 1 elements.
We can insert n in the previously generated permutations right to left then switch direction every time a newpermutation of {1, . . . , n − 1} needs to be processed.
An example of applying this approach bottom up for n = 3

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 bottom up.

Advantages

- Each permutation can be obtained from its immediate predecessor by exchanging just two elements in it.
- The minimal-change requirement is beneficial both for the algorithm's speed and for applications using the permutations.

Disadvatanges

To get the permutations for n elements we need to know the permutations for (n-1) elements

Johnson Trotter algorithm:

Using Johnson Trotter algorithm, the permutations can be achieved 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.

```
ALGORITHM JohnsonTrotter(n)
    //Implements Johnson-Trotter algorithm for generating permutations
    //Input: A positive integer n
    //Output: A list of all permutations of {1, . . . , 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 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
```

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

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

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 Θ(n!).

Lexicographic ordering

Lexicographic order is the order in which they would be listed in a dictionary if the numbers were interpreted as letters of an alphabet. For example, for n = 3,

123 132 213 231 312 321.

```
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
```

2) 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 $A = \{a_1, \ldots, a_n\}$ and all 2^n bit strings $b_1, \ldots, b_n$ 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 ai belongs to the subset and $b_i = 0$ if ai does not belong to it.

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

FIGURE 4.10 Generating subsets bottom up.

For example, for the case of n = 3, we obtain

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