

ANALYSIS AND DESIGN OF ALGORITHMS

SUBJECT CODE: MCA-44

Prepared By:
S.P.Sreeja
Asst. Prof.,
Dept. of MCA
New Horizon College of Engineering

NOTE:

- These notes are intended for use by students in MCA
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:

Text Books:

1. Introduction to Design and Analysis of Algorithms by Anany Levitin, Pearson Edition, 2003.

Reference Books:

1. Introduction to Algorithms by Cormen T.H., Leiserson C. E., and Rivest R. L., PHI, 1998.
2. Computer Algorithms by Horowitz E., Sahani S., and Rajasekharan S., Galgotia Publications, 2001.

Chapter 1: INTRODUCTION.....	1
1.1 : Notion of algorithm.....	1
1.2 : Fundamentals of Algorithmic problem solving.....	3
1.3 : Important problem types.....	6
1.4 : Fundamental data structures.....	9
Chapter 2: FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY.....	18
2.1 : Analysis Framework.....	18
2.2 : Asymptotic Notations and Basic efficiency classes.....	23
2.3 : Mathematical analysis of Non recursive algorithms.....	29
2.4 : Mathematical analysis of recursive algorithms.....	32
2.5 : Examples.....	35
Chapter 3: BRUTE FORCE.....	39
3.1 : Selection sort and Bubble sort.....	39
3.2 : Sequential search Brute Force string matching.....	43
3.3 : Exhaustive search.....	45
Chapter 4: DIVIDE AND CONQUER.....	49
4.1 : Merge sort.....	50
4.2 : Quick sort.....	53
4.3 : Binary search.....	57
4.4 : Binary tree traversals and related properties.....	60
4.5 : Multiplication of large integers	63
4.6 : Strassen's Matrix Multiplication.....	65

Chapter 5: DECREASE AND CONQUER.....	67
5.1 : Insertion sort.....	68
5.2 : Depth – First and Breadth – First search.....	71
5.3 : Topological Sorting.....	77
5.4 : Algorithm for Generating Combinatorial Objects.....	80
Chapter 6: TRANSFORM-AND-CONQUER.....	84
6.1 : Presorting.....	84
6.2 : Balanced Search Trees.....	86
6.3 : Heaps and heap sort.....	92
6.4 : Problem reduction.....	96
Chapter 7: SPACE AND TIME TRADEOFFS.....	98
7.1 : Sorting by counting.....	99
7.2 : Input enhancement in string matching.....	101
7.3 : Hashing.....	106
Chapter 8: DYNAMIC PROGRAMMING.....	109
8.1 : Computing a Binomial Coefficient	109
8.2 : Warshall's and Floyd's Algorithms.....	111
8.3 : The Knapsack problem and Memory functions.....	117
Chapter 9: GREEDY TECHNIQUE.....	121
9.1 : Prim's algorithm.....	121
9.2 : Kruskal's algorithm.....	124
9.3 : Dijkstra's Algorithm.....	129
9.4 : Huffman Trees.....	131

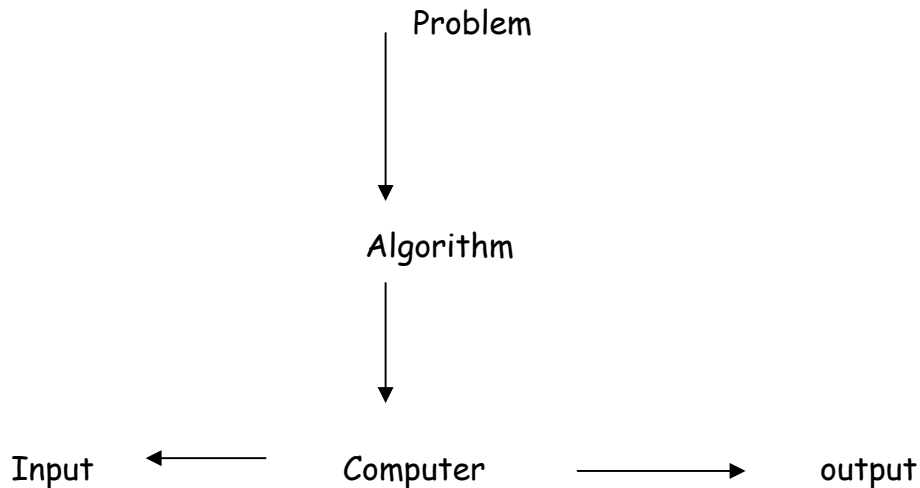
Chapter 10: LIMITATIONS OF ALGORITHM POWER.....	134
10.1: Lower-Bound Arguments.....	134
10.2: Decision Trees.....	134
10.3: P, NP, and NP-complete problems.....	135
Chapter 11: COPING WITH THE LIMITATIONS OF ALGORITHM POWER.....	136
11.1: Backtracking.....	136
11.2: Branch-and-Bound.....	139
11.3: Approximation Algorithms for NP-hard problems.....	143

Analysis and Design of Algorithms

Chapter 1. Introduction

1.1 Notion of algorithm:

An algorithm is a sequence of unambiguous instructions for solving a problem. I.e., for obtaining a required output for any legitimate input in a finite amount of time



There are various methods to solve the same problem.

The important points to be remembered are:

1. The non-ambiguity requirement for each step of an algorithm cannot be compromised.
2. The range of input for which an algorithm works has to be specified carefully.
3. The same algorithm can be represented in different ways.
4. Several algorithms for solving the same problem may exist.
5. Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

The example here is to find the gcd of two integers with three different ways: The gcd of two nonnegative, not-both -zero integers m & n , denoted as $\text{gcd}(m, n)$ is defined as

the largest integer that divides both m & n evenly, i.e., with a remainder of zero. Euclid of Alexandria outlined an algorithm, for solving this problem in one of the volumes of his Elements.

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

is applied repeatedly until $m \bmod n$ is equal to 0;

since $\text{gcd}(m, 0) = m$. {the last value of m is also the gcd of the initial m & n .}

The structured description of this algorithm is:

Step 1: If $n=0$, return the value of m as the answer and stop; otherwise, proceed to step2.

Step 2: Divide m by n and assign the value of the remainder to r .

Step 3: Assign the value of n to m and the value of r to n . Go to step 1.

The psuedocode for this algorithm is:

Algorithm Euclid (m, n)

// Computer gcd (m, n) by Euclid's algorithm.

// Input: Two nonnegative, not-both-zero integers m & n .

//output: gcd of m & n .

While $n \neq 0$ do

$R = m \bmod n$

$m = n$

$n = r$

return m

This algorithm comes to a stop, when the 2nd no becomes 0. The second number of the pair gets smaller with each iteration and it cannot become negative. Indeed, the new value of n on the next iteration is $m \bmod n$, which is always smaller than n . hence, the value of the second number in the pair eventually becomes 0, and the algorithm stops.

Example: $\text{gcd}(60, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12$.

The second method for the same problem is: obtained from the definition itself. i.e., gcd of m & n is the largest integer that divides both numbers evenly. Obviously, that number cannot be greater than the second number (or) smaller of these two numbers,

which we will denote by $t = \min \{m, n\}$. So start checking whether t divides both m and n : if it does t is the answer ; if it doesn't t is decreased by 1 and try again. (Do this repeatedly till you reach 1 and then stop for the example given below)

Consecutive integer checking algorithm:

Step 1: Assign the value of $\min \{m, n\}$ to t .

Step 2: Divide m by t . If the remainder of this division is 0, go to step 3; otherwise go to step 4.

Step 3: Divide n by t . If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to step 4.

Step 4: Decrease the value of t by 1. Go to step 2.

Note : this algorithm, will not work when one of its input is zero. So we have to specify the range of input explicitly and carefully.

The third procedure is as follows:

Step 1: Find the prime factors of m .

Step 2: Find the prime factors of n .

Step 3: Identify all the common factors in the two prime expansions found in step 1 & 2.
(If p is a common factor occurring p_m & p_n times in m and n , respectively, it should be repeated $\min \{ p_m, p_n \}$ times.).

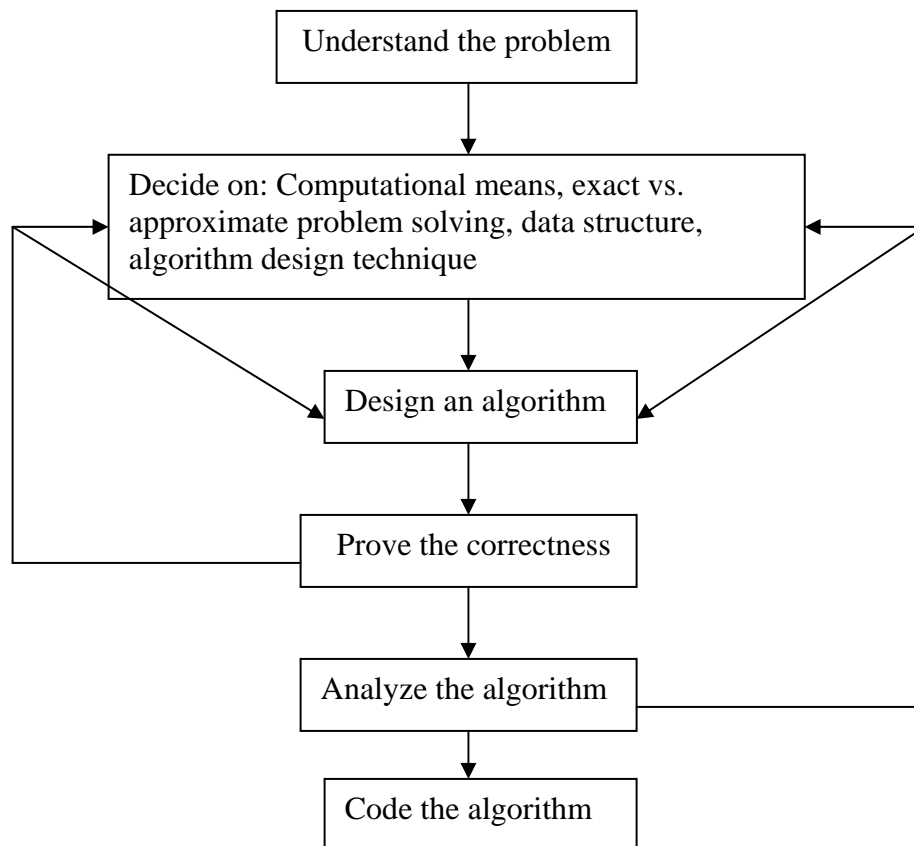
Step 4: Compute the product of all the common factors and return it as gcd of the numbers given.

Example: $60 = 2.2.3.5$
 $24 = 2.2.2.3$
 $\gcd(60, 24) = 2.2.3 = 12$.

This procedure is more complex and ambiguity arises since the prime factorization is not defined. So to make it as an efficient algorithm, incorporate the algorithm to find the prime factors.

1.2 Fundamentals of Algorithmic Problem Solving:

Algorithms can be considered to be procedural solutions to problems. There are certain steps to be followed in designing and analyzing an algorithm.

*** Understanding the problem:**

An input to an algorithm specifies an instance of the problem the algorithm solves. It's also important to specify exactly the range of instances the algorithm needs to handle. Before this we have to clearly understand the problem and clarify the doubts after leading the problems description. Correct algorithm should work for all possible inputs.

***Ascertaining the capabilities of a computational Device:**

The second step is to ascertain the capabilities of a machine. The essence of von-Neumann machines architecture is captured by RAM. Here the instructions are executed one after another, one operation at a time, Algorithms designed to be executed on such machines are called sequential algorithms. An algorithm which has the capability of executing the operations concurrently is called parallel algorithms. RAM model doesn't support this.

*** Choosing between exact and approximate problem solving:**

The next decision is to choose between solving the problem exactly or solving it approximately. Based on this, the algorithms are classified as exact and approximation

algorithms. There are three issues to choose an approximation algorithm. First, there are certain problems like extracting square roots, solving non-linear equations which cannot be solved exactly. Secondly, if the problem is complicated it slows the operations. E.g. traveling salesman problem. Third, this algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

***Deciding on data structures:**

Data structures play a vital role in designing and analyzing the algorithms. Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance.

Algorithm + Data structure = Programs

***Algorithm Design Techniques:**

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. Learning these techniques are important for two reasons, First, they provide guidance for designing for new problems. Second, algorithms are the cornerstones of computer science. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

***Methods of specifying an Algorithm:**

A pseudocode, which is a mixture of a natural language and programming language like constructs. Its usage is similar to algorithm descriptions for writing pseudocode there are some dialects which omits declarations of variables, use indentation to show the scope of the statements such as if, for and while. Use \rightarrow for assignment operations, (//) two slashes for comments.

To specify algorithm flowchart is used which is a method of expressing an algorithm by a collection of connected geometric shapes consisting descriptions of the algorithm's steps.

*** Proving an Algorithm's correctness:**

Correctness has to be proved for every algorithm. To prove that the algorithm gives the required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. A technique used for proving correctness is by mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. But we need one instance of its input for which the algorithm fails. If it is incorrect, redesign the algorithm, with the same decisions of data structures design technique etc.

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithm. For example, in gcd (m,n) two observations are made. One is the second number gets smaller on every iteration and the algorithm stops when the second number becomes 0.

*** Analyzing an algorithm:**

There are two kinds of algorithm efficiency: time and space efficiency. Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs. Another desirable characteristic is simplicity. Simpler algorithms are easier to understand and program, the resulting programs will be easier to debug. For e.g. Euclid's algorithm to find gcd (m,n) is simple than the algorithm which uses the prime factorization. Another desirable characteristic is generality. Two issues here are generality of the problem the algorithm solves and the range of inputs it accepts. The designing of algorithm in general terms is sometimes easier. For eg, the general problem of computing the gcd of two integers and to solve the problem. But at times designing a general algorithm is unnecessary or difficult or even impossible. For eg, it is unnecessary to sort a list of n numbers to find its median, which is its $[n/2]$ th smallest element. As to the range of inputs, we should aim at a range of inputs that is natural for the problem at hand.

*** Coding an algorithm:**

Programming the algorithm by using some programming language. Formal verification is done for small programs. Validity is done thru testing and debugging. Inputs should fall within a range and hence require no verification. Some compilers allow code optimization which can speed up a program by a constant factor whereas a better algorithm can make a difference in their running time. The analysis has to be done in various sets of inputs.

A good algorithm is a result of repeated effort & work. The program's stopping / terminating condition has to be set. The optimality is an interesting issue which relies on the complexity of the problem to be solved. Another important issue is the question of whether or not every problem can be solved by an algorithm. And the last, is to avoid the ambiguity which arises for a complicated algorithm.

1.3 Important problem types:

The two motivating forces for any problem is its practical importance and some specific characteristics. The different types are:

1. Sorting
2. Searching

3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems.

We use these problems to illustrate different algorithm design techniques and methods of algorithm analysis.

1. Sorting:

Sorting problem is one which rearranges the items of a given list in ascending order. We usually sort a list of numbers, characters, strings and records similar to college information about their students, library information and company information is chosen for guiding the sorting technique. For eg in student's information, we can sort it either based on student's register number or by their names. Such pieces of information is called a key. The most important when we use the searching of records. There are different types of sorting algorithms. There are some algorithms that sort an arbitrary of size n using $n \log_2 n$ comparisons, On the other hand, no algorithm that sorts by key comparisons can do better than that. Although some algorithms are better than others, there is no algorithm that would be the best in all situations. Some algorithms are simple but relatively slow while others are faster but more complex. Some are suitable only for lists residing in the fast memory while others can be adapted for sorting large files stored on a disk, and so on.

There are two important properties. The first is called stable, if it preserves the relative order of any two equal elements in its input. For example, if we sort the student list based on their GPA and if two students GPA are the same, then the elements are stored or sorted based on its position. The second is said to be 'in place' if it does not require extra memory. There are some sorting algorithms that are in place and those that are not.

2. Searching:

The searching problem deals with finding a given value, called a search key, in a given set. The searching can be either a straightforward algorithm or binary search algorithm which is a different form. These algorithms play a important role in real-life applications because they are used for storing and retrieving information from large databases. Some algorithms work faster but require more memory, some are very fast but applicable only to sorted arrays. Searching, mainly deals with addition and deletion

of records. In such cases, the data structures and algorithms are chosen to balance among the required set of operations.

3. String processing:

A String is a sequence of characters. It is mainly used in string handling algorithms. Most common ones are text strings, which consists of letters, numbers and special characters. Bit strings consist of zeroes and ones. The most important problem is the string matching, which is used for searching a given word in a text. For e.g. sequential searching and brute-force string matching algorithms.

4. Graph problems:

One of the interesting area in algorithmic is graph algorithms. A graph is a collection of points called vertices which are connected by line segments called edges. Graphs are used for modeling a wide variety of real-life applications such as transportation and communication networks.

It includes graph traversal, shortest-path and topological sorting algorithms. Some graph problems are very hard, only very small instances of the problems can be solved in realistic amount of time even with fastest computers. There are two common problems: the traveling salesman problem, finding the shortest tour through n cities that visits every city exactly once. The graph-coloring problem is to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are of the same color. It arises in event-scheduling problem, where the events are represented by vertices that are connected by an edge if the corresponding events cannot be scheduled in the same time, a solution to this graph gives an optimal schedule.

5. Combinatorial problems:

The traveling salesman problem and the graph-coloring problem are examples of combinatorial problems. These are problems that ask us to find a combinatorial object such as permutation, combination or a subset that satisfies certain constraints and has some desired (e.g. maximizes a value or minimizes a cost).

These problems are difficult to solve for the following facts. First, the number of combinatorial objects grows extremely fast with a problem's size. Second, there are no known algorithms, which are solved in acceptable amount of time.

6. Geometric problems:

Geometric algorithms deal with geometric objects such as points, lines and polygons. It also includes various geometric shapes such as triangles, circles etc. The applications for these algorithms are in computer graphic, robotics etc.

The two problems most widely used are the closest-pair problem, given 'n' points in the plane, find the closest pair among them. The convex-hull problem is to find the smallest convex polygon that would include all the points of a given set.

7. Numerical problems:

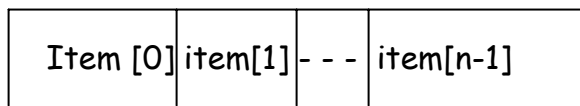
This is another large special area of applications, where the problems involve mathematical objects of continuous nature: solving equations computing definite integrals and evaluating functions and so on. These problems can be solved only approximately. These require real numbers, which can be represented in a computer only approximately. It can also lead to an accumulation of round-off errors. The algorithms designed are mainly used in scientific and engineering applications.

1.4 Fundamental data structures:

Data structure play an important role in designing of algorithms, since it operates on data. A data structure can be defined as a particular scheme of organizing related data items. The data items range from elementary data types to data structures.

*Linear Data structures:

The two most important elementary data structure are the array and the linked list. Array is a sequence contiguously in computer memory and made accessible by specifying a value of the array's index.



Array of n elements.

The index is an integer ranges from 0 to n-1. Each and every element in the array takes the same amount of time to access and also it takes the same amount of computer storage.

Arrays are also used for implementing other data structures. One among is the string: a sequence of alphabets terminated by a null character, which specifies the end of the string. Strings composed of zeroes and ones are called binary strings or bit strings. Operations performed on strings are: to concatenate two strings, to find the length of the string etc.

A linked list is a sequence of zero or more elements called nodes each containing two kinds of information: data and a link called pointers, to other nodes of the linked list. A pointer called null is used to represent no more nodes. In a singly linked list, each node except the last one contains a single pointer to the next element.

item 0	item 1	item n-1	null
--------	--------	-------	----------	------

Singly linked list of n elements.

To access a particular node, we start with the first node and traverse the pointer chain until the particular node is reached. The time needed to access depends on where in the list the element is located. But it doesn't require any reservation of computer memory, insertions and deletions can be made efficiently.

There are various forms of linked list. One is, we can start a linked list with a special node called the header. This contains information about the linked list such as its current length, also a pointer to the first element, a pointer to the last element.

Another form is called the doubly linked list, in which every node, except the first and the last, contains pointers to both its successor and its predecessor.

The another more abstract data structure called a linear list or simply a list. A list is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed are searching for, inserting and deleting on element.

Two special types of lists, stacks and queues. A stack is a list in which insertions and deletions can be made only at one end. This end is called the top. The two operations done are: adding elements to a stack (pushed off). Its used in recursive algorithms, where the last-in- first- out (LIFO) fashion is used. The last inserted will be the first one to be removed.

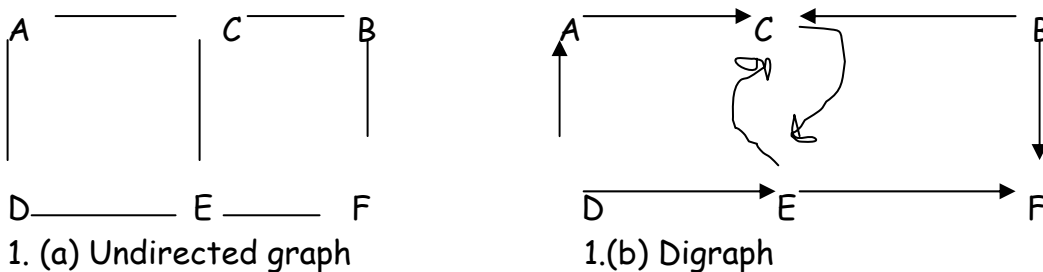
A queue, is a list for, which elements are deleted from one end of the structure, called the front (this operation is called dequeue), and new elements are added to the other end, called the rear (this operation is called enqueue). It operates in a first-in-first-out basis. Its having many applications including the graph problems.

A priority queue is a collection of data items from a totally ordered universe. The principal operations are finding its largest elements, deleting its largest element and adding a new element. A better implementation is based on a data structure called a heap.

***Graphs:**

A graph is informally thought of a collection of points in a plane called vertices or nodes, some of them connected by line segments called edges or arcs. Formally, a graph $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called edges. If these pairs of vertices are unordered, i.e. a pair of vertices (u, v) is same as (v, u) then G is undirected; otherwise, the edge (u, v) , is directed from vertex u to vertex v , the graph G is directed. Directed graphs are also called digraphs.

Vertices are normally labeled with letters / numbers



The 1st graph has 6 vertices and seven edges.

$V = \{a, b, c, d, e, f\}$,
 $E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$

The digraph has four vertices and eight directed edges:

$V = \{a, b, c, d, e, f\}$,
 $E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$

Usually, a graph will not be considered with loops, and it disallows multiple edges between the same vertices. The inequality for the number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops is :

$$0 \leq |E| \leq |V|(|V| - 1) / 2.$$

A graph with every pair of its vertices connected by an edge is called complete. Notation with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called dense; a graph with few edges relative to the number of its vertices is called sparse.

For most of the algorithm to be designed we consider the (i). Graph representation (ii). Weighted graphs and (iii). Paths and cycles.

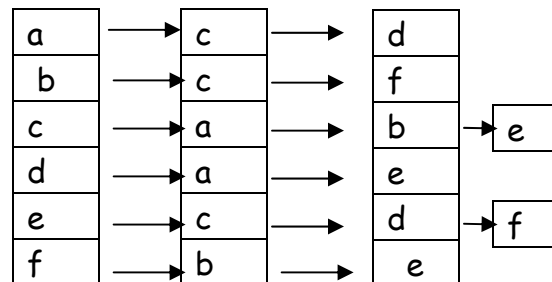
(i) Graph representation:

Graphs for computer algorithms can be represented in two ways: the adjacency matrix and adjacency linked lists. The adjacency matrix of a graph with n vertices is a $n \times n$ Boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i^{th} row and j^{th} column is equal to 1 if there is an edge from the i^{th} vertex to the j^{th} vertex and equal to 0 if there is no such edge. The adjacency matrix for the undirected graph is given below:

Note: The adjacency matrix of an undirected graph is symmetric. i.e. $A[i, j] = A[j, i]$ for all $0 \leq i, j \leq n-1$.

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

1.(c) adjacency matrix



1.(d) adjacency linked list

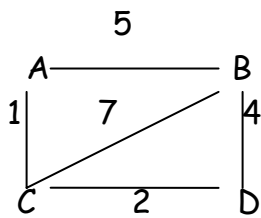
The adjacency linked lists of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the lists vertex. The lists indicate columns of the adjacency matrix that for a given vertex, contain 1's. The lists consumes less space if it's a sparse graph.

(ii) Weighted graphs:

A weighted graph is a graph or digraph with numbers assigned to its edges. These numbers are weights or costs. The real-life applications are traveling salesman problem, Shortest path between two points in a transportation or communication network.

The adjacency matrix. $A[i, j]$ will contain the weight of the edge from the i^{th} vertex to the j^{th} vertex if there exist an edge; else the value will be 0 or ∞ , depends on

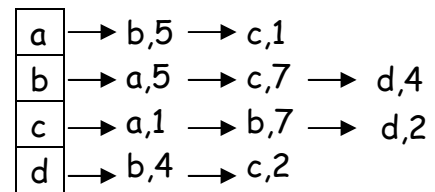
the problem. The adjacency linked list consists of the nodes name and also the weight of the edges.



2(a) weighted graph

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

2(b) adjacency matrix



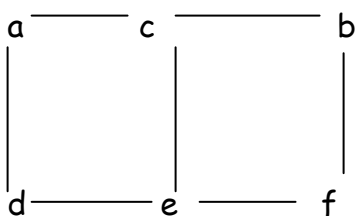
2(c) adjacency linked list

(iii) Paths and cycles:

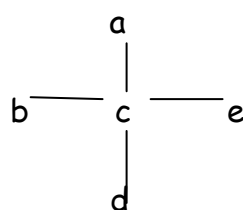
Two properties: Connectivity and acyclicity are important for various applications, which depends on the notion of a path. A path from vertex v to vertex u of a graph G can be defined as a sequence of adjacent vertices that starts with v and ends with u . If all edges of a path are distinct, the path is said to be simple. The length of a path is the total number of vertices in a vertex minus one. For e.g. a, c, b, f is a simple path of length 3 from a to f and a, c, e, c, b, f is a path (not simple) of length 5 from a to f (graph 1.a)

A directed path is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For e.g. a, c, e, f , is a directed path from a to f in the above graph 1. (b).

A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v . If a graph is not connected, it will consist of several connected pieces that are called connected components of the graph. A connected component is the maximal subgraph of a given graph. The graphs (a) and (b) represents connected and not connected graph. For e. g. in (b) there is no path from a to f . it has two connected components with vertices $\{a, b, c, d, e\}$ and $\{f, g, h, i\}$.



3.(a) connected graph

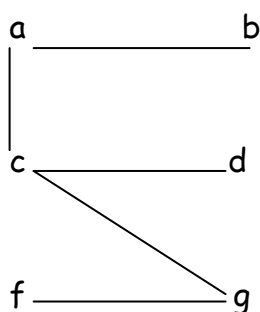


(b) graph that is not connected.

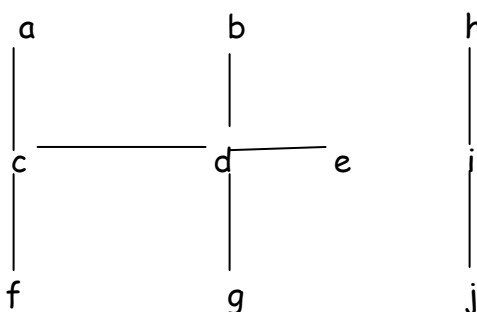
A cycle is a simple path of a positive length that starts and ends at the same vertex. For e.g. f, h, i, g, f is a cycle in graph (b). A graph with no cycles is said to be acyclic.

*** Trees:**

A tree is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a forest: each of its connected components is a tree.



4.(a) Trees



(b) forest

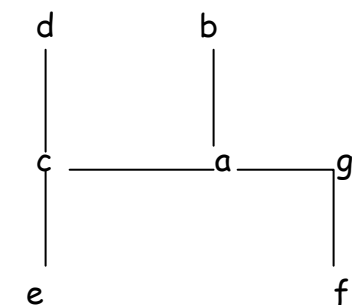
Trees have several important properties :

(i) The number of edges in a tree is always one less than the number of its vertices:

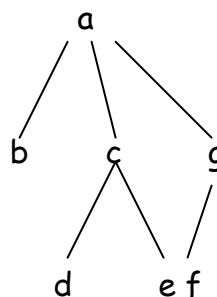
$$|E| = |V| - 1.$$

(ii) Rooted trees:

For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other. For this, select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree. Rooted trees play an important role in various applications with the help of state-space-tree which leads to two important algorithm design techniques: backtracking and branch-and-bound. The root starts from level 0 and the vertices adjacent to the root below is level 1 etc.



(a) free tree



(b) its transformation into a rooted tree

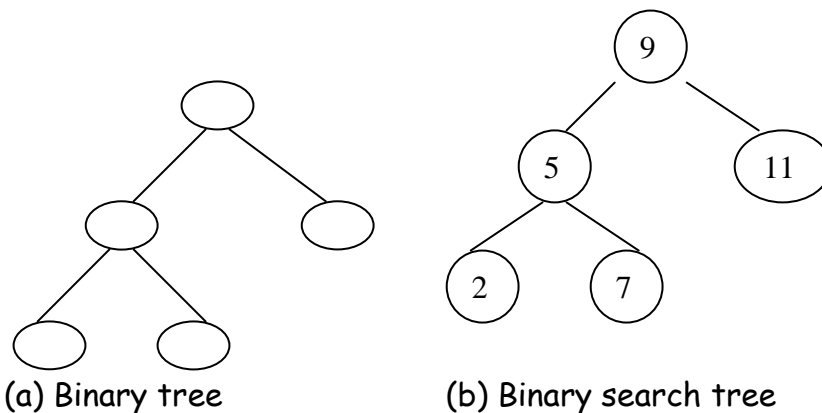
For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called ancestors of V . The set of ancestors that excludes the vertex itself is referred to as proper ancestors. If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq v$), u is said to be the parent of v and v is called a child of u ; vertices that have the same parent are said to be siblings. A vertex with no children is called a leaf; a vertex with at least one child is called parental. All the vertices for which a vertex v is an ancestor are said to be descendants of v . A vertex v with all its descendants is called the sub tree of T rooted at that vertex. For the above tree; a is the root; vertices b, d, e and f are leaves; vertices a, c , and g are parental; the parent of c is a ; the children of c are d and e ; the siblings of b are c and g ; the vertices of the sub tree rooted at c are $\{d, e\}$.

The depth of a vertex v is the length of the simple path from the root to v . The height of the tree is the length of the longest simple path from the root to a leaf. For e.g., the depth of vertex c is 1, and the height of the tree is 2.

(iii) Ordered trees:

An ordered tree is a rooted tree in which all the children of each vertex are ordered. A binary tree can be defined as an ordered tree in which every vertex has no more than two children and each child is a left or right child of its parent. The sub tree with its root at the left (right) child of a vertex is called the left (right) sub tree of that vertex.

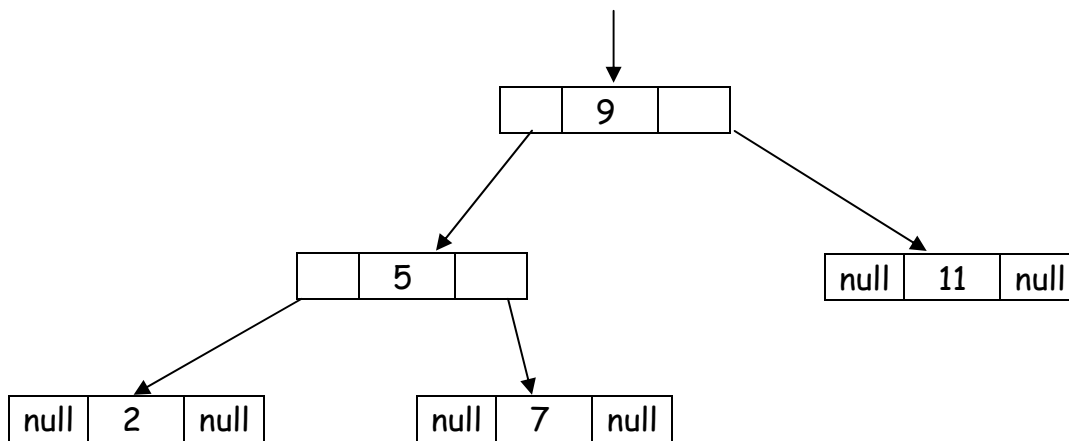
A number assigned to each parental vertex is larger than all the numbers in its left sub tree and smaller than all the numbers in its right sub tree. Such trees are called Binary search trees. Binary search trees can be more generalized to form multiway search trees, for efficient storage of very large files on disks.



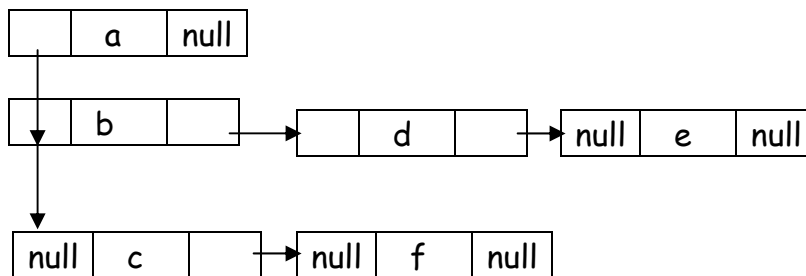
The efficiency of algorithms depends on the tree's height for a binary search tree. Therefore the height h of a binary tree with n nodes follows an inequality for the efficiency of those algorithms:

$$[\log_2 n] \leq h \leq n-1.$$

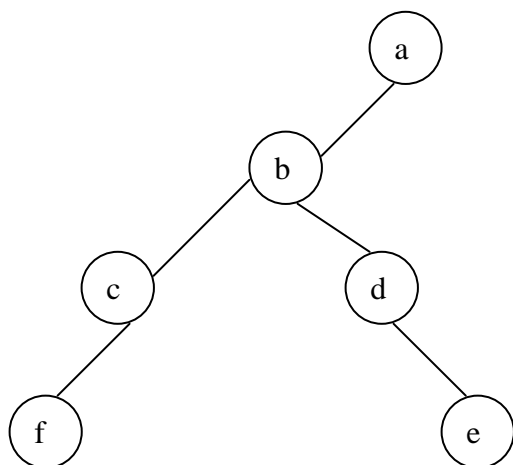
The binary search tree can be represented with the help of linked list: by using just two pointers. The left pointer points to the first child and the right pointer points to the next sibling. This representation is called the first child-next sibling representation. Thus all the siblings of a vertex are linked in a singly linked list, with the first element of the list pointed to by the left pointer of their parent. The ordered tree of this representation can be rotated 45' clock wise to form a binary tree.



Standard implementation of binary search tree (using linked list)



First-child next sibling



Its binary tree representation

* Sets and Dictionaries:

A set can be described as an unordered collection of distinct items called elements of the set. A specific set is defined either by an explicit listing of its elements or by specifying a set of property.

Sets can be implemented in computer applications in two ways. The first considers only sets that are subsets of some large set U called the universal set. If set U has n elements, then any subset S of U can be represented by a bit string of size n , called a bit vector, in which the i^{th} element is 1 iff the i^{th} element of U is included in set S . For e.g. if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ then $S = \{2, 3, 4, 7\}$ can be represented by the bit string as 011010100. Bit string operations are faster but consume a large amount of storage. A multiset or bag is an unordered collection of items that are not necessarily distinct. Note, changing the order of the set elements does not change the set, whereas the list is just opposite. A set cannot contain identical elements, a list can.

The operation that has to be performed in a set is searching for a given item, adding a new item, and deletion of an item from the collection. A data structure that implements these three operations is called the dictionary. A number of applications in computing require a dynamic partition of some n -element set into a collection of disjoint subsets. After initialization, it performs a sequence of union and search operations. This problem is called the set union problem.

These data structure play an important role in algorithms efficiency, which leads to an abstract data type (ADT): a set of abstract objects representing data items with a collection of operations that can be performed on them. Abstract data types are commonly used in object oriented languages, such as C++ and Java, that support abstract data types by means of classes.

Chapter 2. Fundamentals of the Analysis of Algorithm Efficiency.

Introduction:

This chapter deals with analysis of algorithms. The American Heritage Dictionary defines "analysis" as the "separation of an intellectual or substantial whole into its constituent parts for individual study". Algorithm's efficiency is determined with respect to two resources: running time and memory space. Efficiency is studied first in quantitative terms unlike simplicity and generality. Second, give the speed and memory of today's computers, the efficiency consideration is of practical importance.

The algorithm's efficiency is represented in three notations: O ("big oh"), Ω ("big omega") and θ ("big theta"). The mathematical analysis shows the framework systematically applied to analyzing the efficiency of nonrecursive algorithms. The main tool of such an analysis is setting up a sum representing the algorithm's running time and then simplifying the sum by using standard sum manipulation techniques.

2.1 Analysis Framework

For analyzing the efficiency of algorithms the two kinds are time efficiency and space efficiency. Time efficiency indicates how fast an algorithm in question runs; space efficiency deals with the extra space the algorithm requires. The space requirement is not of much concern, because now we have the fast main memory, cache memory etc. so we concentrate more on time efficiency.

- **Measuring an Input's size:**

Almost all algorithms run longer on larger inputs. For example, it takes to sort larger arrays, multiply larger matrices and so on. It is important to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size. For example, it will be the size of the list for problems of sorting, searching etc. For the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n , it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

The size also be influenced by the operations of the algorithm. For e.g., in a spell-check algorithm, it examines individual characters of its input, then we measure the size by the number of characters or words.

Note: measuring size of inputs for algorithms involving properties of numbers. For such algorithms, computer scientists prefer measuring size by the number b of bits in the n 's binary representation.

$$b = \log_2^{n+1}.$$

- **Units for measuring Running time:**

We can use some standard unit of time to measure the running time of a program implementing the algorithm. The drawbacks to such an approach are: the dependence on the speed of a particular computer, the quality of a program implementing the algorithm. The drawback to such an approach are : the dependence on the speed of a particular computer, the quality of a program implementing the algorithm, the compiler used to generate its machine code and the difficulty in clocking the actual running time of the program. Here, we do not consider these extraneous factors for simplicity.

One possible approach is to count the number of times each of the algorithm's operations is executed. The simple way, is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time and compute the number of times the basic operation is executed.

The basic operation is usually the most time consuming operation in the algorithm's inner most loop. For example, most sorting algorithm works by comparing elements (keys), of a list being sorted with each other; for such algorithms, the basic operation is the key comparison.

Let C_{op} be the time of execution of an algorithm's basic operation on a particular computer and let $c(n)$ be the number of times this operations needs to be executed for this algorithm. Then we can estimate the running time, $T(n)$ as:

$$T(n) \sim C_{op} c(n)$$

Here, the count $c(n)$ does not contain any information about operations that are not basic and in tact, the count itself is often computed only approximately. The constant C_{op} is also an approximation whose reliability is not easy to assess. If this algorithm is executed in a machine which is ten times faster than one we have, the running time is also ten times or assuming that $C(n) = \frac{1}{2} n(n-1)$, how much longer will the algorithm run if we double its input size? The answer is four times longer. Indeed, for all but very small values of n ,

$$C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2$$

and therefore,

$$\frac{T(2n)}{T(n)} \approx \frac{C_{op} C(2n)}{C_{op} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}(n)^2} = 4$$

Here C_{op} is unknown, but still we got the result, the value is cancelled out in the ratio. Also, $\frac{1}{2}$ the multiplicative constant is also cancelled out. Therefore, the efficiency analysis framework ignores multiplicative constants and concentrates on the counts' order of growth to within a constant multiple for large size inputs.

- **Orders of Growth:**

This is mainly considered for large input size. On small inputs if there is difference in running time it cannot be treated as efficient one.

Values of several functions important for analysis of algorithms:

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}		

The function growing slowly is the logarithmic function, logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes. Although specific values of such a count depend, of course, in the logarithm's base, the formula

$$\log_a n = \log_a b \times \log_b n$$

Makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant.

On the other end, the exponential function 2^n and the factorial function $n!$ grow so fast even for small values of n . These two functions are required to as exponential-growth functions. "Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes."

Another way to appreciate the qualitative difference among the orders of growth of the functions is to consider how they react to, say, a twofold increase in the value of their argument n . The function $\log_2 n$ increases in value by just 1 (since $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$); the linear function increases twofold; the $n \log n$ increases slightly more than two fold; the quadratic n^2 as fourfold (since $(2n)^2 = 4n^2$) and the cubic function n^3 as eight fold (since $(2n)^3 = 8n^3$); the value of 2^n is squared (since $2^{2n} = (2^n)^2$) and $n!$ increases much more than that.

- **Worst-case, Best-case and Average-case efficiencies:**

The running time not only depends on the input size but also on the specifics of a particular input. Consider the example, sequential search. It's a straightforward algorithm that searches for a given item (search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

The psuedocode is as follows.

```

Algorithm sequential search {A [0..n-1], k}
// Searches for a given value in a given array by Sequential search
// Input: An array A[0..n-1] and a search key K
// Output: Returns the index of the first element of A that matches K or -1 if there is
//         no match
i ← 0
while i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1

```

Clearly, the running time of this algorithm can be quite different for the same list size n . In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n ; $C_{\text{worst}}(n) = n$.

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n , which is an input of size n for which the algorithm runs the longest among all possible inputs of that size. The way to determine is, to analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $c(n)$ among all possible inputs of size n and then compute this worst-case value $C_{\text{worst}}(n)$.

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n , which is an input of size n for which the algorithm runs the fastest among all inputs of that size. First, determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size n . Then ascertain the value of $C(n)$ on the most convenient inputs. For e.g., for the searching with input size n , if the first element equals to a search key, $C_{\text{best}}(n) = 1$.

Neither the best-case nor the worst-case gives the necessary information about an algorithm's behaviour on a typical or random input. This is the information that the average-case efficiency seeks to provide. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n .

Let us consider again sequential search. The standard assumptions are that:

- (a) the probability of a successful search is equal to p ($0 \leq p \leq 1$), and,
- (b) the probability of the first match occurring in the i^{th} position is same for every i .

Accordingly, the probability of the first match occurring in the i^{th} position of the list is p/n for every i , and the no of comparisons is i for a successful search. In case of unsuccessful search, the number of comparisons is n with probability of such a search being $(1-p)$. Therefore,

$$\begin{aligned}
 C_{\text{avg}}(n) &= [1 \cdot p/n + 2 \cdot p/n + \dots \dots i \cdot p/n + \dots \dots n \cdot p/n] + n \cdot (1-p) \\
 &= p/n [1+2+\dots \dots i+\dots \dots +n] + n \cdot (1-p) \\
 &= p/n \cdot [n(n+1)]/2 + n \cdot (1-p) \quad [\text{sum of 1}^{\text{st}} n \text{ natural number formula}] \\
 &= [p(n+1)]/2 + n \cdot (1-p)
 \end{aligned}$$

This general formula yields the answers. For e.g, if $p=1$ (ie., successful), the average number of key comparisons made by sequential search is $(n+1)/2$; ie, the algorithm will inspect, on an average, about half of the list's elements. If $p=0$ (ie., unsuccessful), the average number of key comparisons will be ' n ' because the algorithm will inspect all n elements on all such inputs.

The average-case is better than the worst-case, and it is not the average of both best and worst-cases.

Another type of efficiency is called amortized efficiency. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure. In some situations a single operation can be expensive, but the total time for an entire sequence of such n operations is always better than the worst-case efficiency of that single operation multiplied by n . It is considered in algorithms for finding unions of disjoint sets.

Recaps of Analysis framework:

- 1) Both time and space efficiencies are measured as functions of the algorithm's i/p size.
- 2) Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- 3) The efficiencies of some algorithms may differ significantly for input of the same size. For such algorithms, we need to distinguish between the worst-case, average-case and best-case efficiencies.
- 4) The framework's primary interest lies in the order of growth of the algorithm's running time as its input size goes to infinity.

2.2 Asymptotic Notations and Basic Efficiency classes:

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, we use three notations: O (big oh), Ω (big omega) and θ (big theta). First, we see the informal definitions, in which $t(n)$ and $g(n)$ can be any non negative functions defined on the set of natural numbers. $t(n)$ is the running time of the basic operation, $c(n)$ and $g(n)$ is some function to compare the count with.

Informal Introduction:

$O[g(n)]$ is the set of all functions with a smaller or same order of growth as $g(n)$

Eg: $n \in O(n^2)$, $100n+5 \in O(n^2)$, $1/2n(n-1) \in O(n^2)$.

The first two are linear and have a smaller order of growth than $g(n)=n^2$, while the last one is quadratic and hence has the same order of growth as n^2 . on the other hand,

$n^3 \in (n^2)$, $0.00001 n^3 \notin O(n^2)$, $n^4+n+1 \notin O(n^2)$. The function n^3 and $0.00001 n^3$ are both cubic and have a higher order of growth than n^2 , and so has the fourth-degree polynomial n^4+n+1

The second-notation, $\Omega [g(n)]$ stands for the set of all functions with a larger or same order of growth as $g(n)$. for eg, $n^3 \in \Omega(n^2)$, $1/2n(n-1) \in \Omega(n^2)$, $100n+5 \notin \Omega(n^2)$

Finally, $\theta [g(n)]$ is the set of all functions that have the same order of growth as $g(n)$.

E.g, an^2+bn+c with $a>0$ is in $\theta(n^2)$

- **O-notation:**

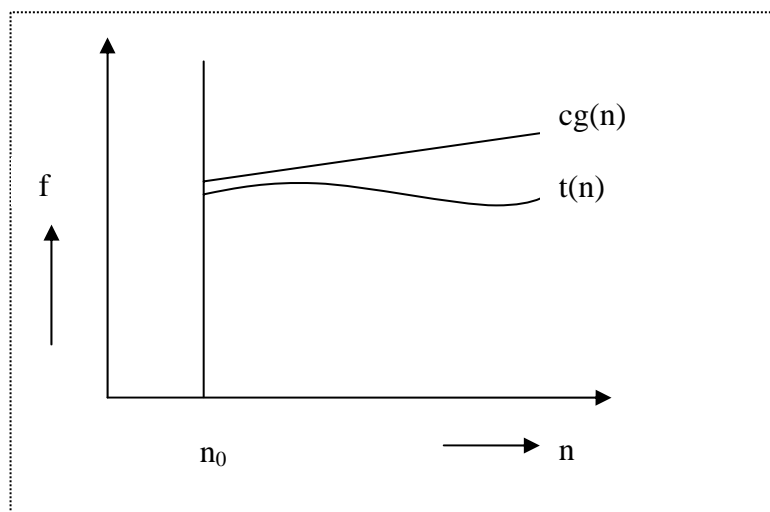
Definition: A function $t(n)$ is said to be in $O[g(n)]$. Denoted $t(n) \in O[g(n)]$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n ie., there exist some positive constant c and some non negative integer n_0 such that $t(n) \leq cg(n)$ for all $n \geq n_0$.

Eg. $100n+5 \in O(n^2)$

Proof: $100n+5 \leq 100n+n$ (for all $n \geq 5$) $= 101n \leq 101 n^2$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5 respectively. The definition says that the c and n_0 can be any value. For eg we can also take. $C = 105$, and $n_0 = 1$.

i.e., $100n+5 \leq 100n+5n$ (for all $n \geq 1$) $= 105n$

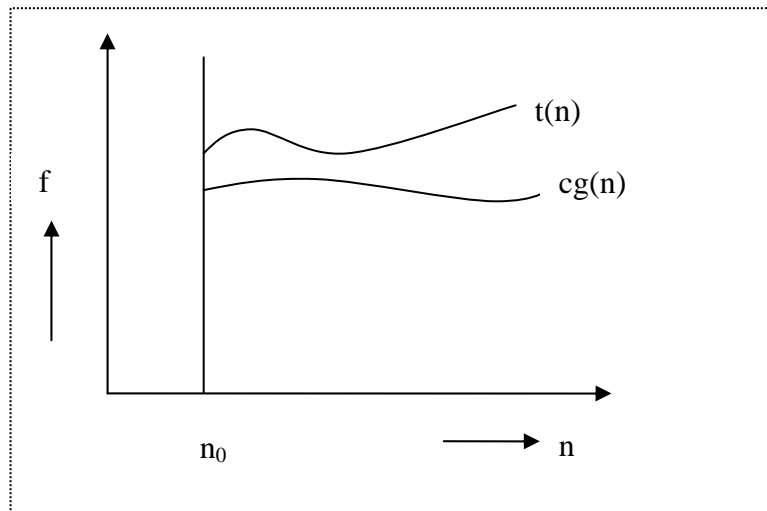


- **Ω -Notation:**

Definition: A fn $t(n)$ is said to be in $\Omega[g(n)]$, denoted $t(n) \in \Omega[g(n)]$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n , i.e., there exist some positive constant c and some non negative integer n_0 s.t.

$$t(n) \geq cg(n) \text{ for all } n \geq n_0.$$

For example: $n^3 \in \Omega(n^2)$, Proof is $n^3 \geq n^2$ for all $n \geq n_0$. i.e., we can select $c=1$ and $n_0=0$.



- **θ - Notation:**

Definition: A function $t(n)$ is said to be in $\theta [g(n)]$, denoted $t(n) \in \theta (g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n , i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that $c_2g(n) \leq t(n) \leq c_1g(n)$ for all $n \geq n_0$.

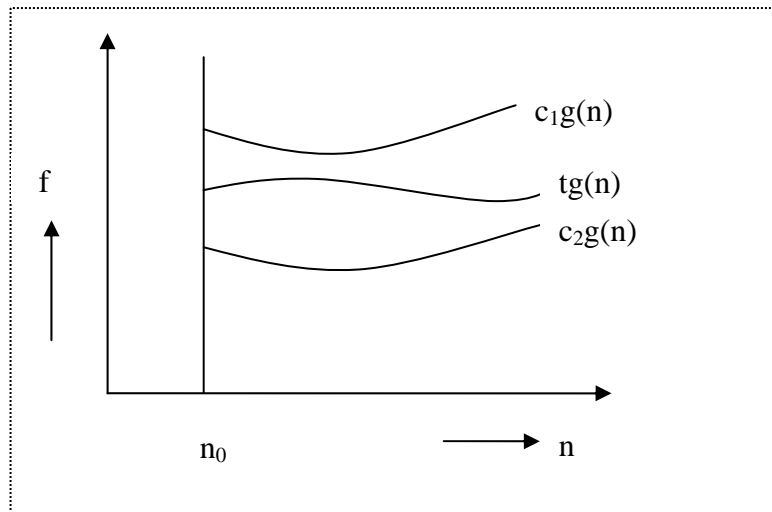
Example: Let us prove that $\frac{1}{2} n(n-1) \in \theta (n^2)$. First, we prove the right inequality (the upper bound)

$$\frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \text{ for all } n \geq n_0.$$

Second, we prove the left inequality (the lower bound)

$$\frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n \cdot \frac{1}{2} n \text{ for all } n \geq 2 = \frac{1}{4} n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$ and $n_0 = 2$



Useful property involving these Notations:

The property is used in analyzing algorithms that consists of two consecutively executed parts:

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

PROOF (As we shall see, the proof will extend to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2 , and b_2 : if $a_1 < b_1$ and $a_2 < b_2$ then $a_1 + a_2 < 2 \max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some constant c and some nonnegative integer n_1 such that

$$t_1(n) < c_1 g_1(n) \text{ for all } n > n_1$$

$$\text{since } t_2(n) \in O(g_2(n)),$$

$$t_2(n) < c_2 g_2(n) \text{ for all } n > n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n > \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &< c_1 g_1(n) + c_2 g_2(n) \\ &< c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \\ &< c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

This implies that the algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part:

$$t_1(n) \in O(g_1(n))$$

$$t_2(n) \in O(g_2(n)) \text{ then } t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

For example, we can check whether an array has identical elements by means of the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $1/2n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n-1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in

$$(O(\max\{n^2, n\}) = O(n^2).$$

Using Limits for Comparing Orders of Growth:

The convenient method for doing the comparison is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = 0 \text{ implies that } t(n) \text{ has a smaller order of growth than } g(n)$$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = c \text{ implies that } t(n) \text{ has the same order of growth as } g(n)$$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \infty \text{ implies that } t(n) \text{ has a larger order of growth than } g(n).$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

EXAMPLE 1 Compare orders of growth of $\frac{1}{2}n(n-1)$ and n^2 . (This is one of the examples we did above to illustrate the definitions.)

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} (1 - 1/n) = \frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Basic Efficiency Classes :

Even though the efficiency analysis framework puts together all the functions whose orders of growth differ by a constant multiple, there are still infinitely many such classes. (For example, the exponential functions a^n have different orders of growth for different values of base a .) Therefore, it may come as a surprise that the time efficiencies

of a large number of algorithms fall into only a few classes. These classes are listed in Table in increasing order of their orders of growth, along with their names and a few comments.

You could raise a concern that classifying algorithms according to their asymptotic efficiency classes has little practical value because the values of multiplicative constants are usually left unspecified. This leaves open a possibility of an algorithm in a worse efficiency class running faster than an algorithm in a better efficiency class for inputs of realistic sizes. For example, if the running time of one algorithm is n^3 while the running time of the other is $10^6 n^2$, the cubic algorithm will outperform the quadratic algorithm unless n exceeds 10^6 . A few such anomalies are indeed known. For example, there exist algorithms for matrix multiplication with a better asymptotic efficiency than the cubic efficiency of the definition-based algorithm (see Section 4.5). Because of their much larger multiplicative constants, however, the value of these more sophisticated algorithms is mostly theoretical.

Fortunately, multiplicative constants usually do not differ that drastically. As a rule, you should expect an algorithm from a better asymptotic efficiency class to outperform an algorithm from a worse class even for moderately sized inputs. This observation is especially true for an algorithm with a better than exponential running time versus an exponential (or worse) algorithm.

Class	Name	Comments
1	Constant	Short of best case efficiency when its input grows the time also grows to infinity.
$\log n$	Logarithmic	It cannot take into account all its input, any algorithm that does so will have atleast linear running time.
n	Linear	Algorithms that scan a list of size n , eg., sequential search
$n \log n$	$n \log n$	Many divide & conquer algorithms including mergesort quicksort fall into this class
n^2	Quadratic	Characterizes with two embedded loops, mostly sorting and matrix operations.
n^3	Cubic	Efficiency of algorithms with three embedded loops,
2^n	Exponential	Algorithms that generate all subsets of an n -element set
$n!$	factorial	Algorithms that generate all permutations of an n -element set

2.3 Mathematical Analysis of Non recursive Algorithms:

In this section, we systematically apply the general framework outlined in Section 2.1 to analyzing the efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing such algorithms.

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is a pseudocode of a standard algorithm for solving the problem.

```
ALGORITHM MaxElement( $A[0..n - 1]$ )
//Determines the value of the largest element in a given array
//Input: An array  $A[0..n - 1]$  of real numbers
//Output: The value of the largest element in  $A$ 
maxval  $\leftarrow A[0]$ 
for  $i \leftarrow 1$  to  $n - 1$  do
  if  $A[i] > \text{maxval}$ 
    maxval  $\leftarrow A[i]$ 
  return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n . The operations that are going to be executed most often are in the algorithm's for loop. There are two operations in the loop's body: the comparison $A[i] > \text{maxval}$ and the assignment $\text{maxval} \leftarrow A[i]$. Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. (Note that the number of comparisons will be the same for all arrays of size n ; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.)

Let us denote $C(n)$ the number of times this comparison is executed and try to find a formula expressing it as a function of size n . The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds between 1 and $n - 1$ (inclusively). Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1$$

This is an easy sum to compute because it is nothing else but 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Here is a general plan to follow in analyzing nonrecursive algorithms.

General Plan for Analyzing Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed-form formula for the count or, at the very least, establish its order of growth.

In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=1}^u c a_i = c \sum_{i=1}^u a_i \quad \text{----(R1)}$$

$$\sum_{i=1}^u (a_i \pm b_i) = \sum_{i=1}^u a_i \pm \sum_{i=1}^u b_i \quad \text{----(R2)}$$

and two summation formulas

$$\sum_{i=l}^u 1 = u - l + 1 \quad \text{where } l \leq u \text{ are some lower and upper integer limits --- (S1)}$$

$$\sum_{i=0}^n i = 1+2+\dots+n = [n(n+1)]/2 \approx \frac{1}{2} n^2 \in \Theta(n^2) \quad \text{----(S2)}$$

(Note that the formula which we used in Example 1, is a special case of formula (S1) for $l = 0$ and $n = n - 1$)

EXAMPLE 2 Consider the *element uniqueness problem*: check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements(A[0..n - 1])*
 //Checks whether all the elements in a given array are distinct
 //Input: An array *A[0..n - 1]*
 //Output: Returns "true" if all the elements in *A* are distinct
 // and "false" otherwise.
 for $i \leftarrow 0$ to $n - 2$ do
 for $j' \leftarrow i + 1$ to $n - 1$ do
 if $A[i] = A[j]$
 return false
 return true

The natural input's size measure here is again the number of elements in the array, i.e., n . Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. Note, however, that the number of element comparisons will depend not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst case input is an array for which the number of element comparisons $C_{\text{worst}}(n)$ is the largest among all arrays of size n . An inspection of the innermost loop reveals that there are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits $i + 1$ and $n - 1$; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and $n - 2$. Accordingly, we get

$$C_{\text{worst}}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - [(n-2)(n-1)]/2$$

$$= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Also it can be solved as (by using S2)

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \dots + 1 = [(n-1)n]/2 \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements.

2.4 Mathematical Analysis of Recursive Algorithms:

In this section, we systematically apply the general framework to analyze the efficiency of recursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing recursive algorithms.

Example 1: Compute the factorial function $F(n) = n!$ for an arbitrary non negative integer n . Since,

$$n! = 1 * 2 * \dots * (n-1) * n = n(n-1)!$$

For $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1).n$ with the following recursive algorithm.

```

ALGORITHM F(n)
// Computes n! recursively
// Input: A nonnegative integer n
// Output: The value of n!
if n = 0 return 1
else return F(n - 1) * n

```

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication, whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0.$$

to compute $F(n-1)$ to multiply $F(n-1)$ by n

Indeed, $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n .

The last equation defines the sequence $M(n)$ that we need to find. Note that the equation defines $M(n)$ not explicitly, i.e., as a function of n , but implicitly as a function of its value at another point, namely $n-1$. Such equations are called recurrence relations or, for

brevity, recurrences. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. Our goal now is to solve the recurrence relation $M(n) = M(n - 1) + 1$, i.e., to find an explicit formula for the sequence $M(n)$ in terms of n only.

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ return 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the code's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications. Thus, the initial condition we are after is

$$M(0) = 0.$$

the calls stop when $n = 0$ ——— ' ——— no multiplications when $n = 0$

Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n - 1) + 1 \text{ for } n > 0, \quad (2.1)$$

$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$F(n) = F(n - 1) \cdot n \text{ for every } n > 0, F(0) = 1.$$

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section. As we just showed, $M(n)$ is defined by recurrence (2.1). And it is recurrence (2.1) that we need to solve now.

Though it is not difficult to "guess" the solution, it will be more useful to arrive at it in a systematic fashion. Among several techniques available for solving recurrence relations, we use what can be called the method of backward substitutions. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$\begin{aligned} M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\ &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\ &= [M(n - 3) + 1] + 2 = M(n - 3) + 3. \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula

for the pattern: $M(n) = M(n - i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n - 1) + 1 = \dots = M(n - i) + i = \dots = M(n - n) + n = n.$$

The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences. Also note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.

The issue of time efficiency is actually not that important for the problem of computing $n!$, however. The function's values get so large so fast that we can realistically compute its values only for very small n 's. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analyzing recursive algorithms.

Generalizing our experience with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.

A General Plan for Analyzing Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

Example 2: the algorithm to find the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *BinRec*(n) : -

//Input: A positive decimal integer n

//Output: The number of binary digits in n 's binary representation

if $n = 1$ return 1

else return *BinRec*($n/2$) + 1

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing $\text{BinRec}(n/2)$ is $A(n/2)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(n/2) + 1 \text{ for } n > 1. \quad (2.2)$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0$$

The presence of $\lfloor n/2 \rfloor$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the **smoothness rule** which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n . (Alternatively, after getting a solution for powers of 2, we can sometimes finetune this solution to get a formula valid for an arbitrary n .) So let us apply this recipe to our recurrence, which for $n = 2^k$ takes the form

$$A(2^k) = A(2^{k-1}) + 1 \text{ for } k > 0, \quad A(2^0) = 0$$

Now backward substitutions encounter no problems:

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 \\ &\dots\dots\dots \\ &= A(2^{k-i}) + i \\ &\dots\dots\dots \\ &= A(2^{k-k}) + k \end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k$$

or, after returning to the original variable $n = 2^k$ and, hence, $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

2.5 Example: Fibonacci numbers

In this section, we consider the Fibonacci numbers, a sequence of numbers as 0, 1, 1, 2, 3, 5, 8, That can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1 \quad \text{----(2.3)}$$

and two initial conditions

$$F(0) = 0, F(1) = 1 \quad \text{----(2.4)}$$

The Fibonacci numbers were introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population. Many more examples of Fibonacci-like numbers have since been discovered in the natural world, and they have even been used in predicting prices of stocks and commodities. There are some interesting applications of the Fibonacci numbers in computer science as well. For example, worst-case inputs for Euclid's algorithm happen to be consecutive elements of the Fibonacci sequence. Our discussion goals are quite limited here, however. First, we find an explicit formula for the n th Fibonacci number $F(n)$, and then we briefly discuss algorithms for computing it.

Explicit Formula for the n th Fibonacci Number

If we try to apply the method of backward substitutions to solve recurrence (2.6), we will fail to get an easily discernible pattern. Instead, let us take advantage of a theorem that describes solutions to a *homogeneous second-order linear recurrence with constant coefficients*

$$ax(n) + bx(n-1) + cx(n-2) = 0, \quad \text{-----(2.5)}$$

where a , b , and c are some fixed real numbers ($a \neq 0$) called the coefficients of the recurrence and $x(n)$ is an unknown sequence to be found. According to this theorem—see Theorem 1 in Appendix B—recurrence (2.5) has an infinite number of solutions that can be obtained by one of the three formulas. Which of the three formulas applies for a particular case depends on the number of real roots of the quadratic equation with the same coefficients as recurrence (2.5):

$$ar^2 + br + c = 0. \quad (2.6)$$

Quite logically, equation (2.6) is called the *characteristic equation* for recurrence (2.5).

Let us apply this theorem to the case of the Fibonacci numbers.

$$F(n) - F(n-1) - F(n-2) = 0. \quad (2.7)$$

Its characteristic equation is

$$r^2 - r - 1 = 0,$$

with the roots

$$r_{1,2} = (1 \pm \sqrt{1-4(-1)}) / 2 = (1 \pm \sqrt{5}) / 2$$

Algorithms for Computing Fibonacci Numbers

Though the Fibonacci numbers have many fascinating properties, we limit our discussion to a few remarks about algorithms for computing them. Actually, the sequence grows so fast that it is the size of the numbers rather than a time-efficient method for computing them that should be of primary concern here. Also, for the sake of simplicity, we consider such operations as additions and multiplications at unit cost in the algorithms

that follow. Since the Fibonacci numbers grow infinitely large (and grow rapidly), a more detailed analysis than the one offered here is warranted. These caveats notwithstanding, the algorithms we outline and their analysis are useful examples for a student of design and analysis of algorithms.

To begin with, we can use recurrence (2.3) and initial condition (2.4) for the obvious recursive algorithm for computing $F(n)$.

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition

//Input: A nonnegative integer n

//Output: The nth Fibonacci number

if $n < 1$ return n

else return $F(n - 1) + F(n - 2)$

Analysis:

The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$. Then the numbers of additions needed for computing $F(n - 1)$ and $F(n - 2)$ are $A(n - 1)$ and $A(n - 2)$, respectively, and the algorithm needs one more addition to compute their sum. Thus, we get the following recurrence for $A(n)$:

$$\begin{aligned} A(n) &= A(n - 1) + A(n - 2) + 1 \text{ for } n > 1, \\ A(0) &= 0, A(1) = 0. \end{aligned} \quad (2.8)$$

The recurrence $A(n) - A(n - 1) - A(n - 2) = 1$ is quite similar to recurrence (2.7) but its right-hand side is not equal to zero. Such recurrences are called *inhomo-geneous recurrences*. There are general techniques for solving inhomogeneous recurrences (see Appendix B or any textbook on discrete mathematics), but for this particular recurrence, a special trick leads to a faster solution. We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

$$\begin{aligned} [A(n) + 1] - [A(n - 1) + 1] - [A(n - 2) + 1] &= 0 \text{ and substituting } B(n) = A(n) + 1: \\ B(n) - B(n - 1) - B(n - 2) &= 0 \\ B(0) = 1, \quad B(1) &= 1. \end{aligned}$$

This homogeneous recurrence can be solved exactly in the same manner as recurrence (2.7) was solved to find an explicit formula for $F(n)$.

We can obtain a much faster algorithm by simply computing the successive elements of the Fibonacci sequence iteratively, as is done in the following algorithm.

ALGORITHM $Fib(n)$

//Computes the nth Fibonacci number iteratively by using its definition

```

//Input: A nonnegative integer  $n$ 
//Output: The  $n$ th Fibonacci number
 $F[0] \leftarrow 0$ ;  $F[1] \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$  do
     $F[i] \leftarrow F[i-1] + F[i-2]$ 
return  $F[n]$ 

```

This algorithm clearly makes $n - 1$ additions. Hence, it is linear as a function of n and "only" exponential as a function of the number of bits b in n 's binary representation. Note that using an extra array for storing all the preceding elements of the Fibonacci sequence can be avoided: storing just two values is necessary to accomplish the task.

The third alternative for computing the n th Fibonacci number lies in using a formula. The efficiency of the algorithm will obviously be determined by the efficiency of an exponentiation algorithm used for computing ϕ^n . If it is done by simply multiplying ϕ by itself $n - 1$ times, the algorithm will be in $\Theta(n) = \Theta(2^b)$. There are faster algorithms for the exponentiation problem. Note also that special care should be exercised in implementing this approach to computing the n th Fibonacci number. Since all its intermediate results are irrational numbers, we would have to make sure that their approximations in the computer are accurate enough so that the final round-off yields a correct result.

Finally, there exists a $\Theta(\log n)$ algorithm for computing the n th Fibonacci number that manipulates only integers. It is based on the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1$$

and an efficient way of computing matrix powers.

Chapter 3. Brute Force

Introduction:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. For e.g. the algorithm to find the gcd of two numbers.

Brute force approach is not an important algorithm design strategy for the following reasons:

- First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. Its used for many elementary but algorithmic tasks such as computing the sum of n numbers, finding the largest element in a list and so on.
- Second, for some problem it yields reasonable algorithms of at least some practical value with no limitation on instance size.
- Third, the expense of designing a more efficient algorithm if few instances to be solved and with acceptable speed for solving it.
- Fourth, even though it is inefficient, it can be used to solve small-instances of a problem.
- Last, it can serve as an important theoretical or educational propose.

3.1 Selection sort and Bubble sort:

The application of Brute-force for the problem of sorting: given a list of n orderable items(eg., numbers characters, strings etc.) , rearrange them in increasing order. The straightforward sorting is based on two algorithms selection and bubble sort. Selection sort is better than the bubble sorting, but, both are better algorithms because of its clarity.

Selection sort:

By scanning the entire list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then scan the list with the second element, to find the smallest among the last $n-1$ elements and exchange it with second element. Continue this process till the $n-2$ elements. Generally, in the i^{th} pass through the list, numbered 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i .

$A_0 \leq A_1 \leq \dots \leq A_{i-1} \mid A_i, \dots, A_{\min}, \dots, A_{n-1}$
 in their final positions the last $n-i$ elements

After $n-1$ passes the list is sorted

```

Algorithm selection sort (A[0..n-1])
// The algorithm sorts a given array
//Input: An array A[0..n-1] of orderable elements
// Output: Array A[0..n-1] sorted increasing order
for i ← 0 to n-2 do
    min ← i
    for j ← i + 1 to n-1 do
        if A[j] < A[min] min ← j
    swap A[i] and A[min]
  
```

The e.g. for the list 89,45,68,90,29,34,17 is

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

Analysis:

The input's size is the no of elements 'n' the algorithms basic operation is the key comparison $A[j] < A[\min]$. The number of times it is executed depends on the array size and it is the summation:

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
 &= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\
 &= \sum_{i=0}^{n-2} (n-1-i)
 \end{aligned}$$

Either compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, it is

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
 &= \sum_{i=0}^{n-2} (n-1-i) \\
 &= [n(n-1)]/2
 \end{aligned}$$

Thus selection sort is a $\theta(n^2)$ algorithm on all inputs. Note, The number of key swaps is only $\theta(n)$, or $n-1$.

Bubble sort:

It is based on the comparisons of adjacent elements and exchanging it. This procedure is done repeatedly and ends up with placing the largest element to the last position. The second pass bubbles up the second largest element and so on after $n-1$ passes, the list is sorted. Pass i ($0 \leq i \leq n-2$) of bubble sort can be represented as:

$$A_0, A_j \longleftrightarrow A_{i+1}, \dots, A_{n-i-1} \quad \Bigg| \quad A_{n-i} \leq, \dots, \dots \leq A_{n-1}$$

in their final positions

Algorithm Bubble sort ($A[0..n-1]$)
 // the algm. Sorts array $A[0..n-1]$
 // Input: An array $A[0..n-1]$ of orderable elements.
 // Output: Array $A[0..n-1]$ sorted in ascending order.
 for $i \leftarrow 0$ to $n-2$ do
 for $j \leftarrow 0$ to $n-2-i$ do
 if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$.

Eg of sorting the list 89, 45, 68, 90, 29

I pass: 89 \leftrightarrow 45 ? 68 90 29
 45 89 \leftrightarrow 68 ? 90 29
 45 68 89 \leftrightarrow 90 ? 29
 45 68 89 90 \leftrightarrow 29
 45 68 89 29 90

II pass: 45 \leftrightarrow 68 ? 89 29 90
 45 68 \leftrightarrow 89 ? 29 90
 45 68 89 \leftrightarrow 29 90
 45 68 29 89 90

III pass: 45 \leftrightarrow 68 ? 29 89 90
 45 68 \leftrightarrow 29 89 90
 45 29 68 89 90
 45 68 29 89 90

IV pass: 45 \leftrightarrow 29 68 89 90
 29 45 68 89 90

Analysis:

The no of key comparisons is the same for all arrays of size n , it is obtained by a sum which is similar to selection sort.

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \\
 &= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]
 \end{aligned}$$

$$\begin{aligned}
 &= \sum_{i=0}^{n-2} (n-1-i) \\
 &= [n(n-1)]/2 \in \Theta(n^2)
 \end{aligned}$$

The no of key swaps depends on the input. The worst case of decreasing arrays, is same as the no of key comparisons:

$$S_{\text{worst}}(n) = C(n) = [n(n-1)]/2 \in \Theta(n^2)$$

3.2 Sequential search and Brute Force string matching:

The two applications of searching based on for a given value in a given list. The first one searches for a given value in a given list. The second deals with the string-matching problem with a given text and a pattern to be searched.

Sequential search:

Here, the algorithm compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). Here is an algorithm with enhanced version: where, we append the search key to the end of the list, the search for the key have to be successful, so eliminate a check for the list's end on each iteration.

Algorithm sequential search($A[0..n]$, K)

// Input: An array A of n elements & search key K .

// Output: The position of the first element in $A[0..n-1]$ whose value is equal to K or -1 if

// no such element is found.

$A[n] \leftarrow k$

$i \leftarrow 0$

While $A[i] \neq K$ do

$i \leftarrow i+1$

if $i < n$ return i

else return -1 .

Another , method is to search in a sorted list. So that the searching can be stopped as soon as the element greater than or equal to the search key is encountered.

Analysis:

The efficiency is determined based on the key comparison

$$(1) C_{\text{worst}}(n) = n.$$

when the algorithm runs the longest among all possible inputs i.e., when the search element is the last element in the list.

$$(2) C_{\text{best}}(n) = 1$$

when the search key is the first element on list.

$$(3) C_{\text{avg}}(n) = (n+1)/2.$$

when the search key is almost in the middle position i.e., half the list will be searched.

Note: its almost similar to the straightforward method with slight variations. It remains to be linear in both average and worst cases.

Brute-force string matching:

The problem is: given a string of n characters called the text and a string of m ($m \leq n$) characters called the pattern, find a substring of the text that matches the pattern. The Brute-force algorithm is to align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match or a mismatching pair is encountered. The pattern is shifted one position to the right and character comparisons are resumed, starting again with the first character of the pattern and its counterpart in the text. The last position in the text that can still be a beginning of a matching substring is $n-m$ (provided that text's positions are from 0 to $n-1$). Beyond that there is no enough characters to match, the algorithm can be stopped.

Algorithm Brute Force string match ($T[0..n-1]$, $P[0..m-1]$)

// Input: An array T [$0..n-1$] of n chars, text

// An array P [$0..m-1$] of m chars , a pattern.

// Output: The position of the first character in the text that starts the first

// matching substring if the search is successful and -1 otherwise.

```

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

```

Example: NOBODY - NOTICE is the text and NOT is the pattern to be searched

N	O	B	O	D	Y	-	N	O	T	I	C	E
N	O	T										
	N	O	T									
		N	O	T								
			N	O	T							
				N	O	T						
					N	O	T					
						N	O	T				
							N	O	T			
								N	O	T		
									N	O	T	

Analysis:

The worst case is to make all m comparisons before shifting the pattern, occurs for $n-m+1$ tries. There in the worst case, the algorithm is in $\theta(nm)$. The average case is when there can be most shift after a very few comparisons and it is to be a linear, i.e., $\theta(n+m) = \theta(n)$.

3.3 Exhaustive search :

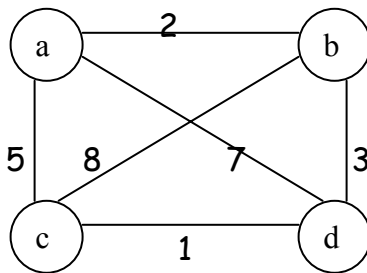
It is a straightforward method used to solve problems of combinatorial problems. It generates each and every element of the problem's domain, selecting based on satisfying the problem's constraints and then finding a desired element(eg., maximization or minimization of desired characteristics). The 3 important problems are TSP, knapsack problem and assignment problem.

* Traveling salesman problem:

The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph- which is a weighted graph, with the graph's vertices representing the cities and the

edge weights specifying the distance. Hamiltonian circuit is defined as a cycle that passes thru all the vertices of the graph exactly once.

The Hamiltonian circuit can also be defined as a sequence of $n+1$ adjacent vertices $v_{i0}, v_{i1}, \dots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one while all other $n-1$ vertices are distinct. Obtain the tours by generating all the permutations of $n-1$ intermediate cities, compute the tour lengths, and find the shortest among them.



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
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$

Consider the condition that the vertex B precedes C then, The total no of permutations will be $(n-1)! / 2$, which is impractical except for small values of n . On the other hand, if the starting vertex is not considered for a single vertex, the number of permutations will be even large for n values.

* Knapsack problem:

The problem states that: given n items of known weights w_1, w_2, \dots, w_n and values v_1, v_2, \dots, v_n and a knapsack of capacity w , find the most valuable subset of the items that fit into the knapsack. Eg consider a transport plane that has to deliver the most valuable set of items to a remote location without exceeding its capacity.

Example:

$W=10$, $w_1, w_2, w_3, w_4 = \{ 7, 3, 4, 5 \}$ and $v_1, v_2, v_3, v_4 = \{ 42, 12, 40, 25 \}$

Subset	Total weight	Total value
\emptyset	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

This problem considers all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets(i.e., the one with the total weight not exceeding the knapsack capacity) and finding the largest among the values, which is an optimal solution. The number of subsets of an n -element set is 2^n the search leads to a $\underline{\Omega}(2^n)$ algorithm, which is not based on the generation of individual subsets.

Thus, for both TSP and knapsack, exhaustive search leads to algorithms that are inefficient on every input. These two problems are the best known examples of NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. The two methods Backtracking and Branch & bound enable us to solve this problem in less than exponential time.

*** Assignment problem:**

The problem is: given n people who need to be assigned to execute n jobs, one person per job. The cost if the i^{th} person is assigned to the j^{th} job is a known quantity

$c[i,j]$ for each pair $i,j=1,2,\dots,n$. The problem is to find an assignment with the smallest total cost.

Example:

	Job1	Job2	Job3	Job4
Person1	9	2	7	8
Person2	6	4	3	7
Person3	5	8	1	8
Person4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \quad \begin{array}{l} \langle 1,2,3,4 \rangle \text{ cost} = 9 + 4 + 1 + 4 = 18 \\ \langle 1,2,4,3 \rangle \text{ cost} = 9 + 4 + 8 + 9 = 30 \\ \langle 1,3,2,4 \rangle \text{ cost} = 9 + 3 + 8 + 4 = 24 \\ \langle 1,3,4,2 \rangle \text{ cost} = 9 + 3 + 8 + 6 = 26 \text{ etc.} \end{array}$$

From the problem, we can obtain a cost matrix, C . The problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

Describe feasible solutions to the assignment problem as n -tuples $\langle j_1, j_2, \dots, j_n \rangle$ in which the i^{th} component indicates the column n of the element selected in the i^{th} row. i.e., The job number assigned to the i^{th} person. For eg $\langle 2,3,4,1 \rangle$ indicates a feasible assignment of person 1 to job2, person 2 to job 3, person3 to job 4 and person 4 to job 1. There is a one-to-one correspondence between feasible assignments and permutations of the first n integers. It requires generating all the permutations of integers $1,2,\dots,n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

Based on number of permutations, the general case for this problem is $n!$, which is impractical except for small instances. There is an efficient algorithm for this problem called the Hungarian method. This problem has a exponential problem solving algorithm, which is also an efficient one. The problem grows exponentially, so there cannot be any polynomial-time algorithm.

Chapter 4. DIVIDE AND CONQUER

Introduction:

Divide and Conquer is a best known design technique, it works according to the following plan:

- a) A problem's instance is divided into several smaller instances of the same problem of equal size.
- b) The smaller instances are solved recursively.
- c) The solutions of the smaller instances are combined to get a solution of the original problem.

As an example, let us consider the problem of computing the sum of n numbers a_0, a_1, \dots, a_{n-1} . If $n > 1$, we can divide the problem into two instances: to compute the sum of first $n/2$ numbers and the remaining $n/2$ numbers, recursively. Once each subset is obtained add the two values to get the final solution. If $n = 1$, then return a_0 as the solution.

$$\text{i.e. } a_0 + a_1 + \dots + a_{n-1} = (a_0 + \dots + a_{n/2-1}) + (a_{n/2} + \dots + a_{n-1})$$

This is not an efficient way, we can use the Brute - force algorithm here. Hence, all the problems are not solved based on divide - and - conquer. It is best suited for parallel computations, in which each sub problem can be solved simultaneously by its own processor.

Analysis:

In general, for any problem, an instance of size n can be divided into several instances of size n/b with a of them needing to be solved. Here, a & b are constants; $a \geq 1$ and $b > 1$. Assuming that size n is a power of b ; to simplify it, the recurrence relation for the running time $T(n)$ is:

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

Where $f(n)$ is a function, which is the time spent on dividing the problem into smaller ones and on combining their solutions. This is called the general divide-and-conquer recurrence. The order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

Theorem:

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the above recurrence equation, then

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

For example, the recurrence equation for the number of additions $A(n)$ made by divide-and-conquer on inputs of size $n=2^k$ is:

$$A(n) = 2 A(n/2) + 1$$

Thus for eg., $a=2$, $b=2$, and $d=0$; hence since $a > b^d$

$$\begin{aligned} A(n) &\in \Theta(n^{\log_b a}) \\ &= \Theta(n^{\log_2 2}) \\ &= \Theta(n^1) \end{aligned}$$

4.1 Merge sort:

It is a perfect example of divide-and-conquer. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0...(n/2-1)]$ and $A[n/2...n-1]$, sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

Algorithm Merge Sort ($A[0..n-1]$)
 //Sorts array A by recursive merge sort
 //Input: An array $A[0..n-1]$ of orderable elements
 //Output: Array $A[0..n-1]$ sorted in increasing order

```

if n>1
    Copy  $A[0...(n/2-1)]$  to  $B[0...(n/2-1)]$ 
    Copy  $A[n/2 ...n-1]$  to  $C[0...(n/2-1)]$ 
    Merge sort ( $B[0..(n/2-1)]$ )
    Merge sort ( $C[0..(n/2-1)]$ )
    Merge ( $B,C,A$ )
  
```

The merging of two sorted arrays can be done as follows: Two pointers are initialized to point to first elements of the arrays being merged. Then the elements pointed to are compared and the smaller of them is added to a new array being constructed; after that, that index of that smaller element is incremented to point to

its immediate successor in the array it was copied from. This operation is continued 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.

Algorithm Merge ($B[0 \dots P-1]$, $C[0 \dots q-1]$, $A[0 \dots p + q-1]$)

//Merge two sorted arrays into one sorted array.

//Input: Arrays $B[0 \dots p-1]$ and $C[0 \dots q-1]$ both sorted

//Output: Sorted Array $A[0 \dots p+q-1]$ of the elements of B & C

$i = 0$; $j = 0$; $k = 0$

while $i < p$ and $j < q$ do

 if $B[i] \leq C[j]$

$A[k] = B[i]$; $i = i+1$

 else

$A[k] = C[j]$; $j = j+1$

$k = k+1$

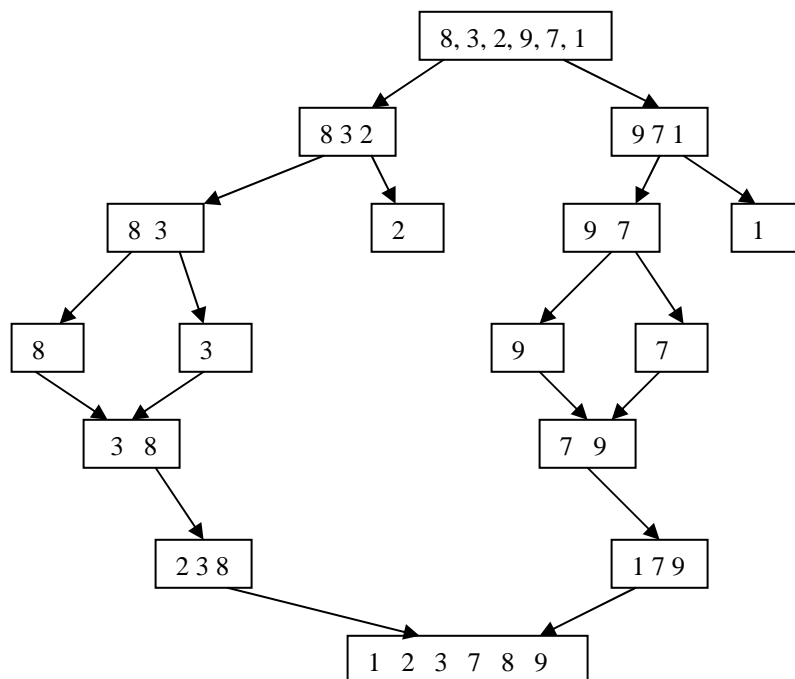
if $i = p$

 copy $C[j \dots q-1]$ to $A[k \dots p+q-1]$

else copy $B[i \dots p-1]$ to $A[k \dots p+q-1]$

Example:

The operation for the list 8, 3, 2, 9, 7, 1 is:



Analysis:

Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2 C(n/2) + C_{\text{merge}}(n) \text{ for } n > 1, c(1) = 0$$

At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one element. In the worst case, neither of the two arrays becomes empty before the other one contains just one element. Therefore, for the worst case, $C_{\text{merge}}(n) = n-1$ and the recurrence is:

$$C_{\text{worst}}(n) = 2C_{\text{worst}}(n/2) + n-1 \text{ for } n > 1, C_{\text{worst}}(1) = 0$$

When n is a power of 2, $n = 2^k$, by successive substitution, we get,

$$\begin{aligned} C(n) &= 2 C(n/2) + Cn \\ &= 2 (2 C(n/4) + C n/2) + Cn \\ &= 2 C(n/4) + 2 Cn \\ &= 4 (2 C(n/8) + C n/4) + 2 Cn \\ &= 8 C(n/8) + 3 Cn \\ &\vdots \\ &\vdots \\ &= 2^k C(1) + k Cn \\ &= an + Cn \log_2 n \end{aligned}$$

Since $k = \log n$ and $n = 2^k$, we get, $\log_2 n = k(\log_2 2) = k * 1$

It is easy to see that if $2^k \leq n \leq 2^{k+1}$, then

$$C(n) \leq C(2^{k+1}) \in \Theta(n \log_2 n)$$

There are 2 inefficiency in this algorithm:

1. It uses 2n locations. The additional n locations can be eliminated by introducing a key field which is a linked field which consists of less space. i.e., LINK (1:n) which consists of [0:n]. These are pointers to elements A. It ends with zero. Consider Q&R, Q=2 and R=5 denotes the start of each lists:

LINK:	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
	6	4	7	1	3	0	8	0

Q = (2,4,1,6) and R = (5,3,7,8)

From this we conclude that $A(2) \leq A(4) \leq A(1) \leq A(6)$ and $A(5) \leq A(3) \leq A(7) \leq A(8)$.

2. The stack space used for recursion. The maximum depth of the stack is proportional to $\log_2 n$. This is developed in top-down manner. The need for stack space can be eliminated if we develop algorithm in Bottom-up approach.

It can be done as an in-place algorithm, which is more complicated and has a larger multiplicative constant.

4.2 Quick Sort:

Quick sort is another sorting algorithm that is based on divide-and-conquer strategy. Quick sort divides according to their values. It rearranges elements of a given array $A[0..n-1]$ to achieve its partition, a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all elements after s are greater than or equal to $A[s]$:

$A[0] \dots A[s-1]$	$[s]$	$A[s+1] \dots A[n-1]$
All are $\leq A[s]$		all are $\geq A[s]$

After this partition $A[s]$ will be in its final position and this proceeds for the two sub arrays:

```

Algorithm Quicksort(A[l..r])
//Sorts a sub array by quick sort
//I/P: A sub array A [l..r] of A [0..n-1], designed by its left and right indices l & r
//O/P: A [l..r] is increasing order - sub array

```

```

if l < r
S ← partition (A[l..r]) // S is a split position
Quick sort A [l...s-1]
Quick sort A [s+1...r]

```

The partition of $A[0..n-1]$ and its sub arrays $A[l..r]$ ($0 \leq l < r \leq n-1$) can be achieved by the following algorithms. First, select an element with respect to whose value we are going to divide the sub array, called as pivot. The pivot by default is considered to be the first element in the list. i.e. $P = A[l]$

The method which we use to rearrange is as follows which is an efficient method based on two scans of the sub array ; one is left to right and the other right to left comparing each element with the pivot. The $L \rightarrow R$ scan starts with the second element. Since we need elements smaller than the pivot to be in the first part of the sub array, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The $R \rightarrow L$ scan starts with last element of the sub array. Since we want elements larger than the pivot to be in the second part of the sub array, this scan skips over elements that are larger than the pivot and stops on encountering the first smaller element than the pivot.

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. $i < j$, exchange $A[i]$ and $A[j]$ and resume the scans by incrementing and decrementing j , respectively.

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

Finally, if the scanning indices stop while pointing to the same elements, i.e. $i = j$, the value they are pointing to must be equal to p . Thus, the array is partitioned. The cases where, $i > j$ and $i = j$ can be combined to have $i \geq j$ and do the exchanging with the pivot.

Algorithm partition ($A[l..r]$)

// Partitions a sub array by using its first elt as a pivot.

// Input: A sub array $A[l..r]$ of $A[0..n-1]$ defined by its left and right indices l & r ($l < r$).

// O/P : A partition of $A[l..r]$ with the split position returned as this function's value.

```

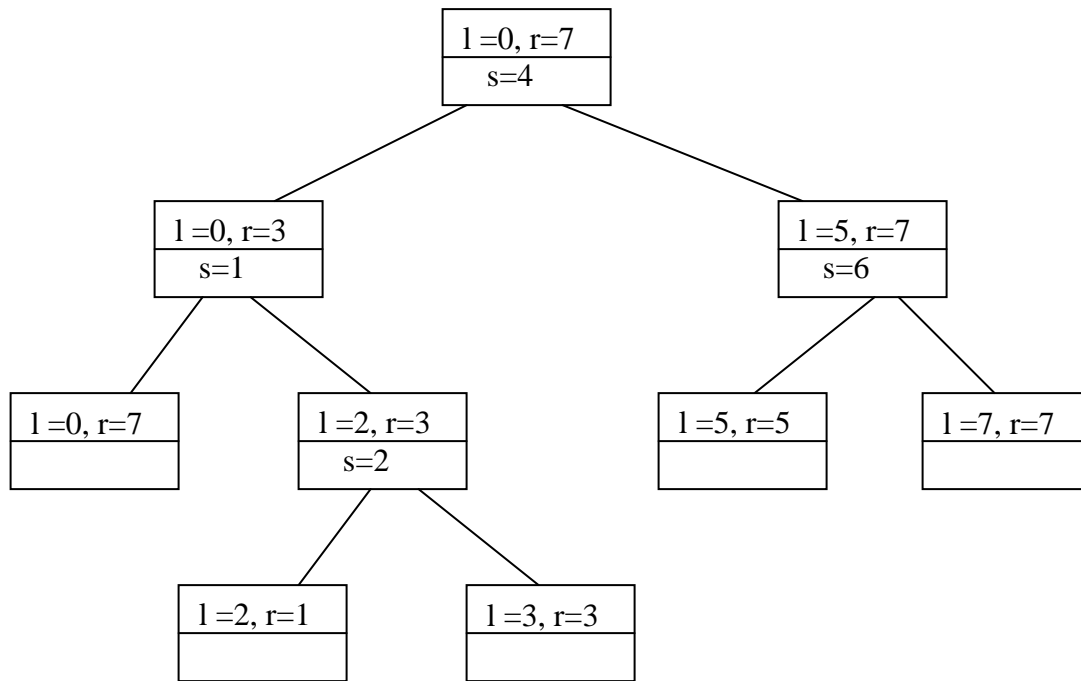
P ← A[l]
i ← l ; j ← r + 1
repeat
    repeat i ← i+1 until A[i] ≥ P
    repeat j ← j-1 until A[j] ≤ P
    swap (A[i], A[j])
until i ≥ j
Swap (A[i], A[j]) //undo the last swap when i ≥ j
Swap (A[l], A[j])
return j

```

Example:

0	1	2	3	4	5	6	7
5	3 ⁱ	1	9	8	2	4	7 ^j
5	3	1	9 ⁱ	8	2	4 ^j	7
5	3	1	4 ⁱ	8	2	9 ^j	7
5	3	1	4	8 ⁱ	2 ^j	9	7
5	3	1	4	2 ^j	8 ⁱ	9	7
5	3	1	4	5	8	9	7
2	3 ⁱ	1	4		8	9 ⁱ	7 ^j
2	3 ⁱ	1	4		8	7 ⁱ	9 ^j
2	1 ^j	3 ⁱ	4		8	7 ^j	9 ⁱ
1	2	3	4		7	8	9
		3	4 ^{ij}		7		
		3	4 ⁱ				9
			4				

Ordered as: 1 2 3 4 5 7 8 9

**Analysis:**

The efficiency is based on the number of key comparisons. If all the splits happen in the middle of the sub arrays, we will have the best case. The no. of key comparisons will be:

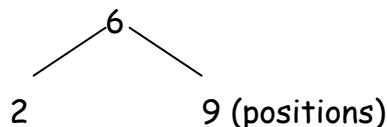
$$C_{\text{best}}(n) = 2 C_{\text{best}}(n/2) + n \text{ for } n > 1$$

$$C_{\text{best}}(1) = 0$$

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

$$C_{\text{best}}(n) = n \log_2 n.$$

In the worst case, all the splits will be skewed to the extreme : one of the two sub arrays will be empty while the size of the other will be just one less than the size of a subarray being partitioned. It happens for increasing arrays, i.e., the inputs which are already solved. If $A[0 \dots n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the $L \rightarrow R$ scan will stop on $A[1]$ while the $R \rightarrow L$ scan will go all the way to reach $A[0]$, indicating the split at position 0:



So, after making $n+1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will find itself with the strictly increasing array $A[1..n-1]$ to sort. This sorting of increasing arrays of diminishing sizes will continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to:

$$C_{\text{worst}}(n) = (n+1) + n + \dots + 3 = \frac{(n+1)(n+2)}{2} - 3$$

$$\in \Theta(n^2)$$

Finally, the average case efficiency, let $C_{\text{avg}}(n)$ be the average number of key comparisons made by quick sort on a randomly ordered array of size n . Assuming that the partition split can happen in each position s ($0 \leq s \leq n-1$) with the same probability $1/n$, we get the following recurrence relation:

$$C_{\text{avg}}(n) = \sum_{s=0}^{n-1} [(n+1) + C_{\text{avg}}(s) + C_{\text{avg}}(n-1-s)]$$

$$C_{\text{avg}}(0) = 0, C_{\text{avg}}(1) = 0$$

$$\text{Therefore, } C_{\text{avg}}(n) \approx 2n \ln 2 \approx 1.38n \log_2 n$$

Thus, on the average, quick sort makes only 38% more comparisons than in the best case. To refine this algorithm : efforts were taken for better pivot selection methods (such as the median - of - three partitioning that uses as a pivot the median of the left most, right most and the middle element of the array) ; switching to a simpler sort on smaller sub files ; and recursion elimination (so called non recursive quick sort). These improvements can cut the running time of the algorithm by 20% to 25%

Partitioning can be useful in applications other than sorting, which is used in selection problem also.

4.3 Binary Search:

It is an efficient algorithm for searching in a sorted array. It works by comparing a search key k with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if $K < A[m]$ and for the second half if $K > A[m]$.

$A[0] \dots A[m-1]$	K $A[m]$	$A[m+1] \dots A[n-1]$
Search here if $K < A[m]$		Search here if $K > A[m]$

As an eg., let us apply binary search to searching for $K = 55$ in the array:

3	14	27	31	42	55	70	81	98
---	----	----	----	----	----	----	----	----

The iterations of the algorithm are given as:

Index	0	1	2	3	4	5	6	7	8
value									
	3	14	27	31	42	55	70	81	98

Itern1			m		r
itern2			m		r

$A[m] = K$, so the algorithm stops

Binary search can also implemented as a nonrecursive algorithm.

Algorithm Binarysearch($A[0..n-1], k$)

```
// Implements nonrecursive binarysearch
```

```
// Input: An array A[0..n-1] sorted in ascending order and a search key k
```

```
// Output: An index of the array's element that is equal to k or -1 if there is no such
```

```
// element
```

$$l \leftarrow 0; r \leftarrow n-1$$

```
while l ≤ r do
```

$$m \leftarrow \lceil (l+r)/2 \rceil$$

```
if k = A[m] return m
```

else if $k < A[m]$ $r \leftarrow m-1$

else $l \leftarrow m+1$

```
return -1
```

Analysis:

The efficiency of binary search is to count the number of times the search key is compared with an element of the array. For simplicity, we consider three-way comparisons. This assumes that after one comparison of K with $A[M]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[M]$. The comparisons not only depends on 'n' but also the particular instance of the problem. The worst case comparison $C_w(n)$ include all arrays that do not contain a search key, after one comparison the algorithm considers the half size of the array.

$$C_w(n) = C_w(n/2) + 1 \text{ for } n > 1, C_w(1) = 1 \quad \text{———— eqn (1)}$$

To solve such recurrence equations, assume that $n = 2^k$ to obtain the solution.

$$C_w(2^k) = k + 1 = \log_2 n + 1$$

For any positive even number n , $n = 2i$, where $i > 0$. now the LHS of eqn (1) is:

$$\begin{aligned} C_w(n) &= [\log_2 n] + 1 = [\log_2 2i] + 1 = [\log_2 2 + \log_2 i] + 1 \\ &= (1 + [\log_2 i]) + 1 = [\log_2 i] + 2 \end{aligned}$$

The R.H.S. of equation (1) for $n = 2i$ is

$$\begin{aligned} C_w[n/2] + 1 &= C_w[2i / 2] + 1 \\ &= C_w(i) + 1 \\ &= ([\log_2 i] + 1) + 1 \\ &= ([\log_2 i] + 2) \end{aligned}$$

Since both expressions are the same, we proved the assertion.

The worst - case efficiency is in $\Theta(\log n)$ since the algorithm reduces the size of the array remained as about half the size, the numbers of iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Also the logarithmic functions grows so slowly that its values remain small even for very large values of n .

The average-case efficiency is the number of key comparisons made which is slightly smaller than the worst case.

$$\text{i.e. } C_{\text{avg}}(n) \approx \log_2 n$$

More accurately, for successful search $C_{\text{avg}}(n) \approx \log_2 n - 1$ and for unsuccessful search $C_{\text{avg}}(n) \approx \log_2 n + 1$.

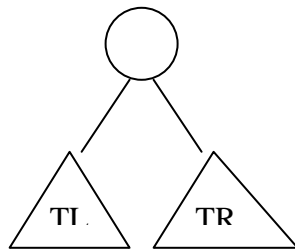
Though binary search is an optional algorithm there are certain algorithms like interpolation search which gives better average - case efficiency. The hashing technique does not even require the array to be sorted. The application of binary search is used for solving non-linear equations in one unknown.

Binary search is sometimes presented as a quint essential example of divide-and-conquer. Because, according to the general technique the problem has to be divided

into several smaller subproblems and then combine the solutions of smaller instances to obtain the original solution. But here, we divide into two subproblems but only one of them need to be solved. So the binary search can be considered as a degenerative case of this technique and it will be more suited for decrease-by-half algorithms.

4.4 Binary Tree Traversals and Related Properties:

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called the left and right sub tree of the root.



Since, here we consider the divide-and-conquer technique that is dividing a tree into left subtree and right subtrees. As an example, we consider a recursive algorithm for computing the height of a binary tree. Note, the height of a tree is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the heights of the root's left and right sub trees plus 1. Also define, the height of the empty tree as - 1. Recursive algorithm is as follows:

Algorithm Height (T)

// Computes recursively the height of a binary tree T

// Input : A binary tree T

// Output : The height of T

if $T = \emptyset$ return - 1

else return $\max \{ \text{Height}(T_L), \text{Height}(T_R) \} + 1$

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T . The number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. The recurrence relation for $A(n(T))$ is:

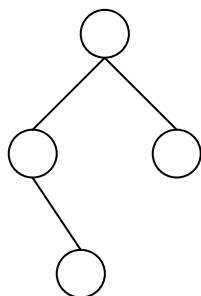
$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \text{ for } n(T) > 0,$$

$$A(0) = 0$$

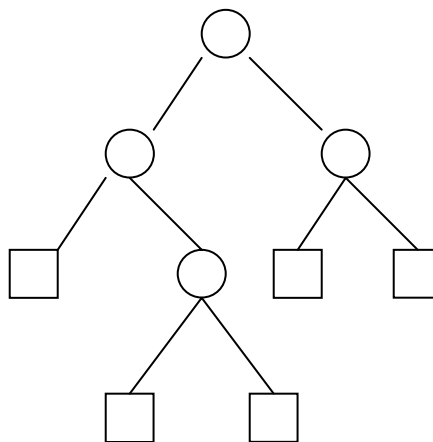
Analysis:

The efficiency is based on the comparisons and addition operations, and also we should check whether the tree is empty or not. For an empty tree, the comparison $T = \emptyset$ is executed once but there are no additions and for a single node tree, the comparison and addition numbers are three and one respectively.

The tree's extension can be drawn by replacing the empty sub trees by special nodes which helps in analysis. The extra nodes (square) are called external; the original nodes (circles) are called internal nodes. The extension of the empty binary tree is a single external node.



Binary tree



Extended form of Binary tree

The algorithm height makes one addition for every internal node of the extended tree and one comparison to check whether the tree is empty for every internal and external node. The number of external nodes x is always one more than the number internal nodes n :

$$\text{i.e. } x = n + 1$$

To prove this equality by induction in the no. of internal nodes $n \geq 0$. The induction's basis is true because for $n = 0$ we have the empty tree with 1 external node by definition: In general, let us assume that

$$x = K + 1$$

for any extended binary tree with $0 \leq K \leq n$ internal nodes. Let T be an extended binary tree with n internal nodes and x external nodes, let n_L and x_L be the number of internal and external nodes in the left sub tree of T and n_R & x_R is the internal & external nodes

of right subtree respectively. Since $n > 0$, T has a root which is its internal node and hence $n = n_L + n_R + 1$ using the equality

$$\begin{aligned}x &= x_L + x_R = (n_L + 1) + (n_R + 1) \\&= (n_L + n_R + 1) + 1 = n + 1\end{aligned}$$

which completes the proof

In algorithm Height, $C(n)$, the number of comparisons to check whether the tree is empty is:

$$\begin{aligned}c(n) &= n + x \\&= n + (n + 1) \\&= 2n + 1\end{aligned}$$

while the number of additions is:

$$A(n) = n$$

The important example is the traversals of a binary tree: Pre order, Post order and In order. All three traversals visit nodes of a binary tree recursively, i.e. by visiting the tree's root and its left and right sub trees. They differ by which the root is visited.

In the Pre order traversal, the root is visited before the left and right subtrees are visited.

In the in order traversal, the root is visited after visiting the left and rightsub trees.

In the Post order traversal, the root is visited after visiting the left and right subtrees.

The algorithm can be designed based on recursive calls. Not all the operations require the traversals of a binary tree. For example, the find, insert and delete operations of a binary search tree requires traversing only one of the two sub trees. Hence, they should be considered as the applications of variable - size decrease technique (Decrease-and-Conquer) but not the divide-and-conquer technique.

4.5 Multiplication of Large Integers:

Some applications like cryptology require manipulation of integers of more than 100 decimal digits. It is too long to fit into a word. Moreover, the multiplication of two n digit numbers, result with n^2 digit multiplications. Here by using divide-and-conquer concept we can decrease the number of multiplications by slightly increasing the number of additions.

For example, say two numbers 23 and 14. It can be represented as follows:

$$23 = 2 \times 10^1 + 3 \times 10^0 \text{ and } 14 = 1 \times 10^1 + 4 \times 10^0$$

Now, let us multiply them:

$$\begin{aligned} 23 * 14 &= (2 \times 10^1 + 3 \times 10^0) * (1 \times 10^1 + 4 \times 10^0) \\ &= (2*1) 10^2 + (3*1 + 2*4) 10^1 + (3*4) 10^0 \\ &= 322 \end{aligned}$$

When done in straight forward it uses the four digit multiplications. It can be reduced to one digit multiplication by taking the advantage of the products $(2 * 1)$ and $(3 * 4)$ that need to be computed anyway.

$$\text{i.e. } 3 * 1 + 2 * 4 + (2 + 3) * (1 + 4) - (2 * 1) - (3 * 4)$$

In general, for any pair of two-digit numbers $a = a_1 a_0$ and $b = b_1 b_0$ their product c can be computed by the formula:

$$C = a * b = C_2 10^2 + C_1 10^1 + C_0$$

Where,

$$C_2 = a_1 * b_1 - \text{Product of 1}^{\text{st}} \text{ digits}$$

$$C_0 = a_0 * b_0 - \text{Product of their 2}^{\text{nd}} \text{ digits}$$

$$C_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0) \text{ product of the sum of the a's digits and the sum of the b's digits minus the sum of } C_2 \text{ and } C_0.$$

Now based on divide-and-conquer, if there are two n -digits integers a and b where n is a positive even number. Divide both numbers in the middle; the first half and second half ' a ' are a_1 and a_0 respectively. Similarly, for b it is b_1 and b_0 respectively.

$$\text{Here, } a = a_1 a_0 \text{ implies that } a = a_1 10^{n/2} + a_0$$

$$\text{And, } b = b_1 b_0 \text{ implies that } b = b_1 10^{n/2} + b_0$$

$$\begin{aligned}
 \text{Therefore, } C &= a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
 &= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0) \\
 &= C_2 10^n + C_1 10^{n/2} + C_0
 \end{aligned}$$

Where, $C_2 = a_1 * b_1$ - Product of first half

$C_0 = a_0 * b_0$ - Product of second half

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0)$ product of the sum of the a's halves and the sum of the b's halves minus the sum of C_2 & C_0 .

If $n/2$ is even, we can apply the same method for computing the products C_2 , C_0 , and C_1 . Thus, if n is power of 2, we can design a recursive algorithm, where the algorithm terminates when $n=1$ or when n is so small that we can multiply it directly.

Analysis:

It is based on the number of multiplications. Since multiplication of n -digit numbers require three multiplications of $n/2$ digit numbers, the recurrence for the number of multiplications $M(n)$ will be

$$M(n) = 3M(n/2) \text{ for } n > 1, M(1) = 1$$

Solving it by backward substitutions for $n=2^k$ yields.

$$\begin{aligned}
 M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2}) \\
 &= \dots 3^i M(2^{k-i}) = \dots 3^k M(2^{k-k}) = 3^k
 \end{aligned}$$

Since $k = \log_2 n$,

$$\begin{aligned}
 M(n) &= 3^{\log_2 n} \\
 &= n^{\log_2 3} \\
 &\approx n^{1.585}
 \end{aligned}$$

$n^{1.585}$, is much less than n^2

4.6 Strassen's Matrix Multiplication:

This is a problem which is used for multiplying two $n \times n$ matrixes. Volker Strassen in 1969 introduced a set of formula with fewer number of multiplications by increasing the number of additions.

Based on straight forward or Brute-Force algorithm.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{bmatrix}$$

But according to strassen's formula's the product of two $n \times n$ matrixes are obtained as:

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

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-by-2 matrixes, Strassen's algorithm requires seven multiplications and 18 additions / subtractions, where as the brute-force algorithm requires eight multiplications and 4 additions. Let A and B be two n -by- n matrixes when n is a power of two. (If not, pad the rows and columns with zeroes). We can divide A, B and their product C into four $n/2$ by $n/2$ sub matrixes as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

Analysis:

The efficiency of this algorithm, $M(n)$ is the number of multiplications in multiplying two n by n matrices according to Strassen's algorithm. The recurrence relation is as follows:

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

Solving it by backward substitutions for $n = 2^k$ yields.

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

Since $k = \log_2 n$,

$$\begin{aligned} M(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &\approx n^{2.807} \end{aligned}$$

which is smaller than n^3 required by Brute force algorithm.

Since this saving is obtained by increasing the number of additions, $A(n)$ has to be checked for obtaining the number of additions. To multiply two matrixes of order $n > 1$, the algorithm needs to multiply seven matrixes of order $n/2$ and make 18 additions of matrixes of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied.

The recurrence relation is

$$\begin{aligned} A(n) &= 7A(n/2) + 18(n/2)^2 \text{ for } n > 1 \\ A(1) &= 0 \end{aligned}$$

This can be deduced based on Master's Theorem, as $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. Thus in Strassen's algorithm it is $\Theta(n^{\log_2 7})$, which is better than $\Theta(n^3)$ of brute force.

Chapter 5. DECREASE AND CONQUER

INTRODUCTION:

The Decrease-and-Conquer technique is 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. It can be established either top down (recursively) or bottom up (nonrecursively). The three major forms are:

- Decrease - by - a constant
- Decrease - by - a constant factor
- Variable size decrease
- Decrease-by-a-constant:

In this, the size of an instance is reduced by the same constant, equal to one, on each iteration of the algorithm. In some cases, the constant will be equal to two, where you have odd and even sizes.

Consider, as an example, the exponentiation problem of computing a^n for positive integer exponents. The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the formula: $a^n = a^{n-1} \times a$. So, the function $f(n) = a^n$ can be computed either top down by using recursive function / definition as:

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

or bottom up multiplying 'a' itself by $n-1$ times.

- Decrease-by-a-constant factor

It reduces a problems instance by the same constant factor on each iteration of the algorithm, mostly by two.

For example, the exponentiation problem with the instance of size n is to compute a^n , the instance of half its size will be to compute $a^{n/2}$, with a relationship between the two: $a^n = (a^{n/2})^2$. This is applicable only for even size of n . If n is odd, we have to compute a^{n-1} by using the same and multiply the result by a . In general, the function is:

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

The algorithm's efficiency is measured by the number of multiplications, which should be in $O(\log n)$ because, on each iteration the size is reduced by at least on half at the expense of no more than two multiplications.

Note: In Divide-and-Conquer the function to solve exponentiation problem is:

$$a^n = \begin{cases} a^{n/2} * a^{n/2} & \text{if } n > 1 \\ a & \text{if } n=1 \end{cases}$$

which is less sufficient compared to the decrease by a constant factor.

- **Variable-size-decrease:**

Here, the size reduction pattern varies from one iteration of an algorithm to another. For example, in Euclid's algorithm for computing the greatest common divisor is designed based on the formula:

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

Though the arguments on the right hand side are always smaller than those on the left hand side, they are smaller neither by a constant nor by a constant factor.

5.1 Insertion Sort:

This sorting is done based on decrease-by-one factor to sort an array $A[0 \dots n-1]$. Here we assume that the smaller problem of sorting the array $A[0 \dots n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \dots A[n-2]$. Here all we need to do is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there.

There are three ways to do this:

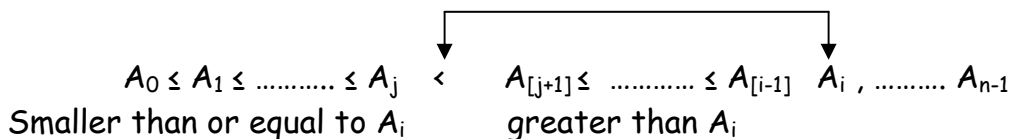
- First, we can scan the sorted sub array from left to right until the first element greater than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right before that element.

- Second, we can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right after that element.

Note: Both are equivalent, usually it is the second one that is implemented in practice it is better for sorted arrays. The resulting algorithm is called straight insertion sort or insertion sort.

- The third way is to use binary search to find an appropriate position for $A[n-1]$ in the sorted portion of the array. The resulting algorithm is called binary insertion sort.

Though insertion sort, is based on a recursive idea, it is more efficient to implement this algorithm bottom-up, i.e., iteratively.



The list with $A[0..n-1]$, it starts with $A[0]$ and ends with $A[n-1]$, $A[i]$ is inserted at its appropriate place among the first i elements of the array that have been already sorted.

Algorithm Insertionsort ($A[0..n-1]$)

// The algorithm sorts a given array

//Input: An array $A[0..n-1]$ of n orderable elements

// Output: Array $A[0..n-1]$ sorted in increasing order

for $i \leftarrow 1$ to $n-1$ do

$v \leftarrow A[i]$

$j \leftarrow i - 1$

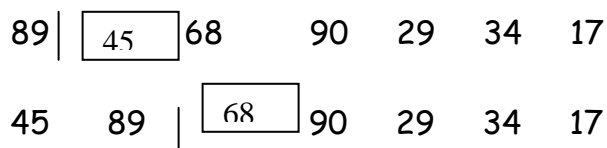
 while $j \geq 0$ and $A[j] > v$ do

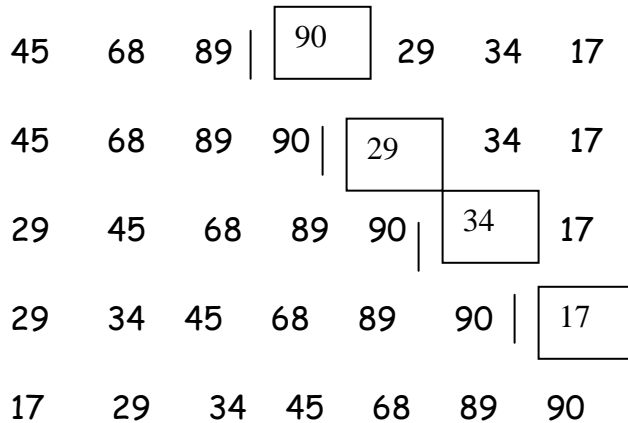
$A[j+1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j+1] \leftarrow v$

The e.g. for the list 89,45,68,90,29,34,17 is



**Analysis:**

The basic operation of the algorithm is the key comparison $A[j] > v$. The number of key comparison in this algorithm 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 $j=i-1, \dots, 0$. Since $V = A[i]$ it happens if and only if $A[j] > A[i]$ for $j=i-1, \dots, 0$. Thus, for the worse case input, we get $A[0] > A[1]$ (for $i=1$), $A[1] > A[2]$ (for $i=2$)....., $A[n-2] > A[n-1]$ (for $i=n-1$). In other words, the worst case input is an array of decreasing values. The number of key comparisons for such an input is:

$$\begin{aligned}
 C_{\text{worst}}(n) &= \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 \\
 &= \sum_{i=1}^{n-1} i \\
 &= [n(n-1)]/2 \in \theta(n^2)
 \end{aligned}$$

- 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 $i=1, \dots, n-1$, i.e. if the input array is already sorted in ascending order. Thus, the number of key comparisons is:

$$\begin{aligned}
 C_{\text{best}}(n) &= \sum_{i=1}^{n-1} 1 = n-1 \in \theta(n)
 \end{aligned}$$

- In the average-case efficiency is based on investigating the number of element pairs that are out of order. It shows that on randomly ordered arrays this sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{\text{avg}}(n) \approx n^2/4 \in \theta(n^2)$$

This makes the algorithm an efficient one.

5.2 Depth – First and Breadth – First search:

This is an important graph algorithms based on decrease-by-one technique. Graphs are interesting data structures with a wide variety of applications. It requires the processing vertices or edges of a graph in a systematic fashion. There are two principal algorithms for doing such traversals; Depth – First search (DFS) and Breadth First search (BFS). Its useful in important properties of a graph investigations.

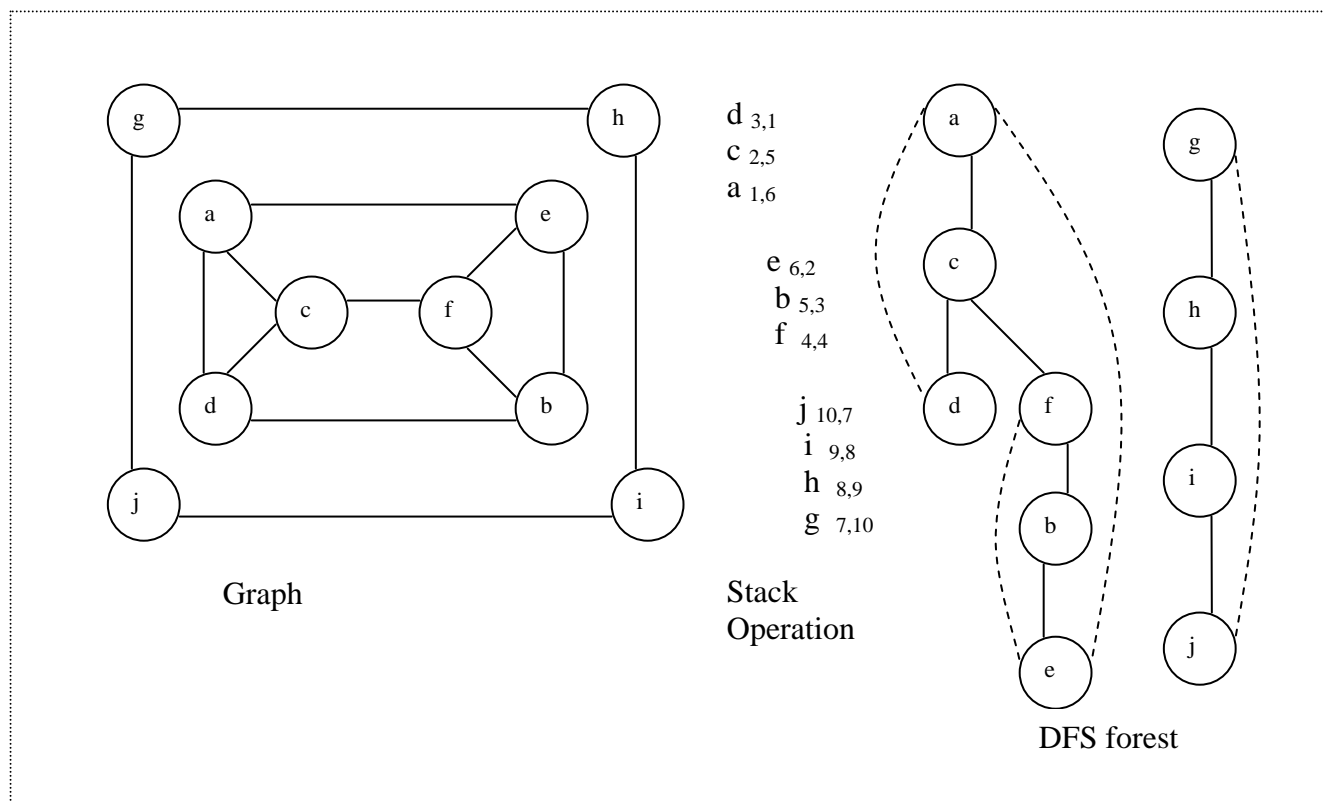
- **Depth – First Search:**

DFS starts visiting vertices of a graph at an arbitrary vertex by making it as having been 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 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 DFS must be restarted at any of them.

It is convenient to use a stack to trace the operation of DFS. 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.

It is also useful in constructing a depth-first search forest. The traversals staring vertex 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, i.e. its parent in the tree. Such an edge is called a back edge because it connects a vertex to its ancestor, other than the parent, in the DFS forest.



Algorithm DFs (G)

// Input : Graph $G = \langle V, E \rangle$

// Output : Graph G with its vertices marked with consecutive integers in the order

// they have been first encountered by the DFS traversal.

Mark each vertex in V with 0 as a marks of being "unvisited".

Count $\leftarrow 0$

for each vertex in v in V do

if v is marked with 0

 dfs (v)

dfs (V)

// visits recursively all the unvisited vertices connected to vertex v and assigns them

// the numbers in the ordering they are encountered via global variable count.

```

count ← count + 1;
mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w).

```

Analysis:

It is quite efficient since it takes just the time proportional to the size of the data structure used for representing the graph. It is better to convert the graph into its adjacency matrix or adjacency linked lists before traversal. Thus, for the adjacency matrix representation, the traversal's time efficiency is in $\Theta(|V|^2)$ and for the adjacency linked lists, it is in $\Theta(|V| + |E|)$ where $|V|$ and $|E|$ are the numbers of the graph's vertices and edges respectively.

Here, tree edges and back edges can be treated as two different classes. Tree edges, are edges used by the DFS to reach previously unvisited vertices. Back edges, connect vertices to previously visited vertices other than their immediate predecessors. They connect vertices their ancestors in the forest other than their parents.

The DFS yields two orderings of vertices the pushing of vertices on to the stack - vertices which reached first and popped off the stack - vertices which become dead ends.

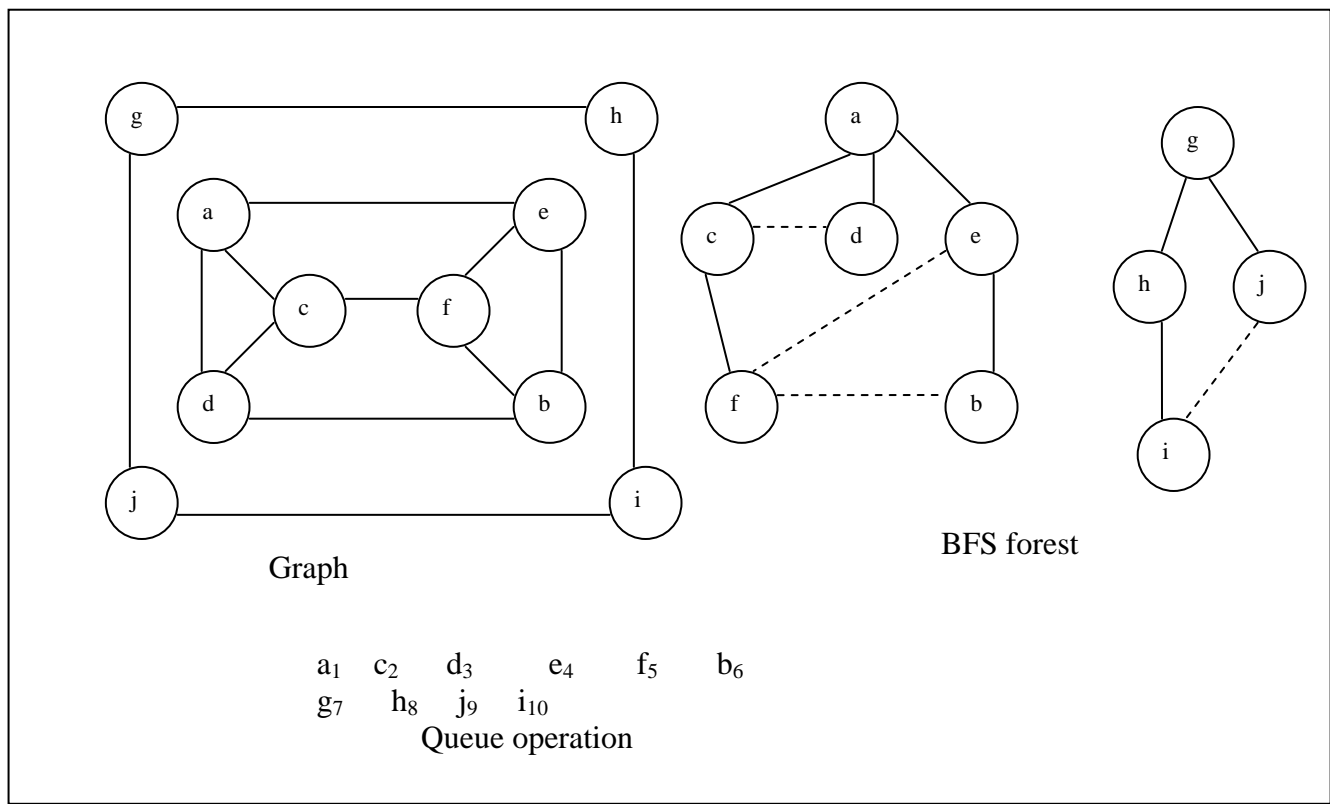
Important elementary applications of DFS include checking connectivity and acyclicity of a graph. Since a DFS halts after visiting all the vertices connected by a path to the starting vertex, checking a graph's connectivity can be done as follows: Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the graph's vertices will have been visited. If they have, the graph is connected; otherwise it is not connected.

For checking acyclicity we can represent the graph in the form of a DFS forest. If it does not have back edges, the graph is acyclic. If there is a back edge from some vertex u to its ancestor v (i.e. back edge from d to a or e to a) the graph is cyclic from u to v .

- **Breadth - First Search:**

BFS proceeds 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.

To trace the operation it is better to use a queue. 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.



It's useful to construct a BFS 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.

Algorithm BFS (G)

// 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)

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 \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's vertex v do

 if w is marked with 0

 count \leftarrow count + 1; mark w with count

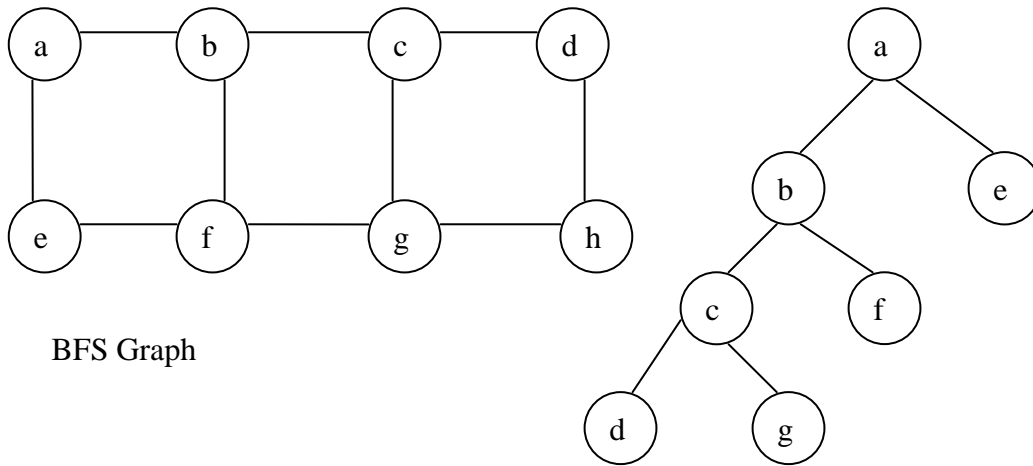
 add w to the queue.

 remove vertex v from the front of the queue.

Analysis:

It has the same efficiency as DFS : it is in $\Theta(|V|^2)$ for adjacency matrix and in $\Theta(|V| + |E|)$ for adjacency linked list representation. It yields a single ordering of vertices because the queue is a FIFO structure where the order of addition and removal of the vertices are same. Two kinds of edges exist ; tree and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connects vertices to those vertices visited before, but they connect either siblings or cousins on the same or adjacent levels of a BFS tree.

Two checkings can be done as DFS i.e., connectivity and acyclicity of a graph. It can be also used for finding a path with the fewest number of edges between two given vertices. We start a BFS traversal at one of the two vertices given and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought.



Part of BFS tree to identify the minimum edge path from a to g

For example, path a-b-c-g in the graph has the fewest number of edges among all the paths between vertices a and g.

Comparison of DFS and BFS:

	DFS	BFS
Data Structure	Stack	Queue
No. of Vertex Orderings	2 Orderings	1 Ordering
Edge Type (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 Linked List	$\Theta(V + E)$	$\Theta(V + E)$

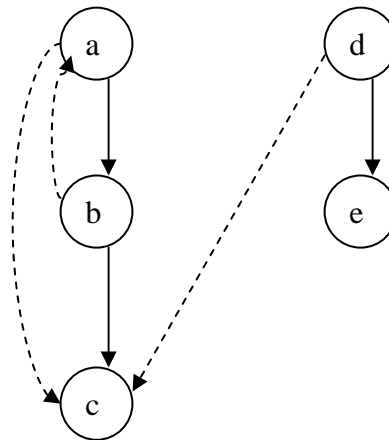
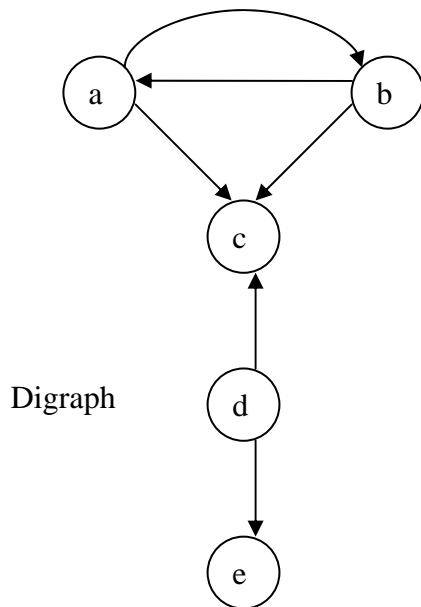
Note: Articulation point → removal of the edges breaks the graph into pieces.

5.3 Topological Sorting:

A directed graph or digraph is a graph with direction specified for all its edges. The two principal means of representing a digraph is the adjacency matrix and linked list. The two differences between the directed and undirected graph in representing them are:

1. The adjacency matrix of a directed graph does not have to be symmetric;
2. An edge in a directed graph has just one corresponding node in the digraph's adjacency linked list.

DFS and BFS are principal traversal algorithm for traversing digraphs, but the structure of corresponding forests can be more complex.



DFS forest of the digraph for the DFS traversal started at vertex a

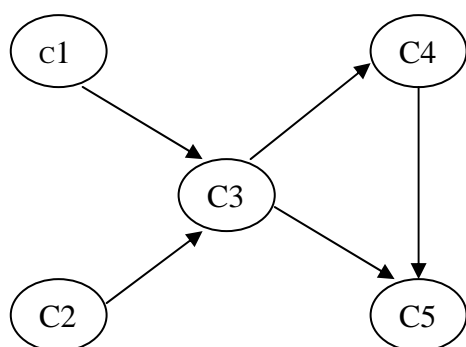
For the digraph, the DFS forest exhibits all four types of edges possible in a DFS forest of a digraph; tree edges (ab,bc,de) back edges (ba) from vertices to the ancestors, forward edges (ac) from vertices to the descendants in the tree other than their children and cross edges (dc), which are none of the other three.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not, the presence of a back edge indicates that the digraph has a directed cycle. A directed cycle in a digraph is a sequence of its vertices that

starts and ends at the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. Inversely, if a DFS forest of a digraph has no back edges the digraph is a directed acyclic graph (dag).

As an example, for the directed graphs, we 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 based on some prerequisites: 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.

The order the courses should be taken can be modeled by a graph in which vertices represent courses and directed edges indicate prerequisite requirements.



Digraph

C5₁
C4₂
C3₃
C1₄
C2₅

Stack
Popping
Off
order

The popping-off order:

C5, C4, C3, C1, C2

The topological sorted list:

C2 → C1 → C3 → C4 → C5

Solution to the problem

The question is, whether we can 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. This problem is called topological sorting. The problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting, a digraph must be a dag. A dag is not only necessary but also sufficient to make sorting possible; i.e. if a digraph has no cycles, the sorting has a solution. There are two efficient algorithms that both verify whether a digraph is a dag, and if it is, produce an ordering of vertices that solves the sorting problem.

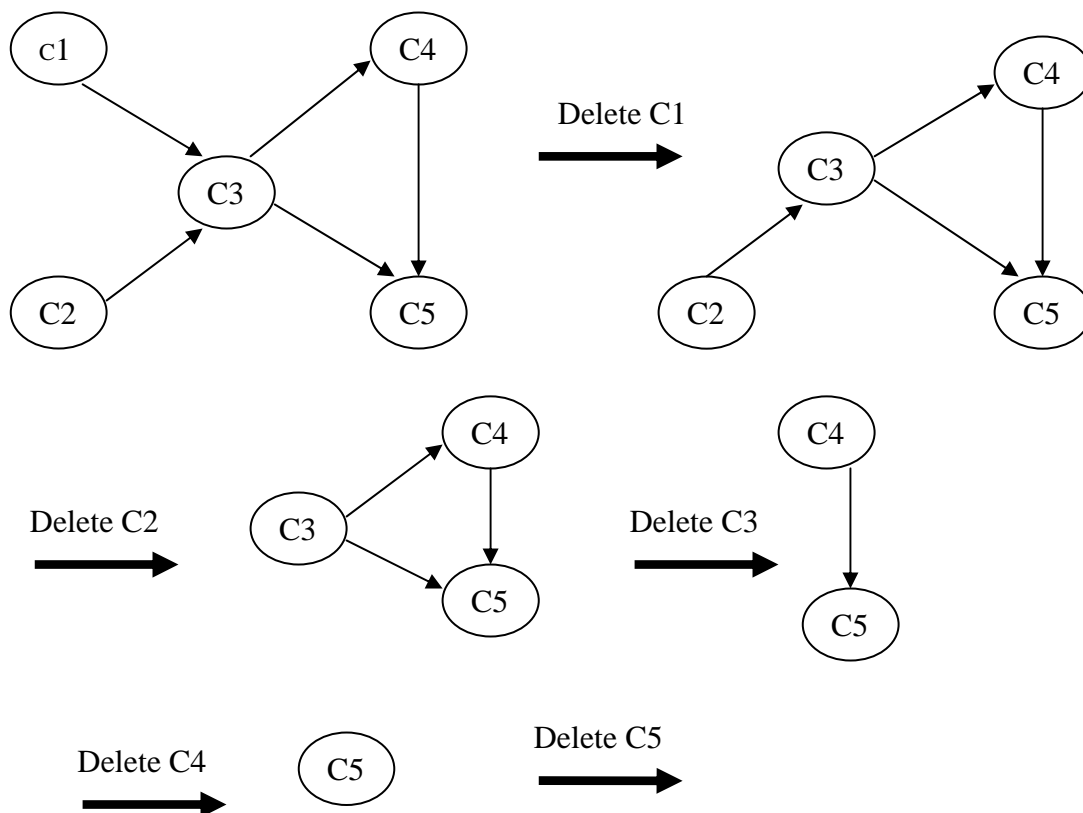
- ❖ The first algorithm is a simple application of DFS; perform a DFS traversal and note that order in which vertices become dead ends (i.e., are popped off from the

stack). Reversing this give a solution. No back edge is encountered in the figure above and if it exist, then the digraph is not a dag and its impossible to obtain the solution.

The algorithm works in such a way that, when a vertex V is popped off a DFS stack, no vertex with an edge from u to v can be among the vertices popped off before v . (Otherwise, (u,v) would have been a back edge). Hence, any such vertex u will be listed after v in the popped-off order list and before v in the reversed list.

- ❖ The second algorithm is based on a direct implementation of the decrease-by-one 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. The order in which the vertices are deleted yields a solution to the problem.

NOTE: If there are several sources, break the tie arbitrarily. If there is none, stop because the problem cannot be solved.



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

The topological sorting problem is used in large projects - that involves more number of interrelated tasks with known prerequisites. The first thing to be ensured is to check whether the prerequisites are contradictory. The convenient method is to create a digraph and solve the problem. The other algorithms are based on CPM and PERT methodologies.

5.4 Algorithm for Generating Combinatorial Objects:

The most important types of combinatorial objects are permutations, combinations and subsets of a given set. This is a branch of discrete mathematics called combinatorics.

- **Generating Permutations:**

For simplicity, we assume that set whose elements need to be permuted is the set of integers 1 to n ; More generally, it can be interpreted as indices of elements in an n -element set $[a_1, \dots, a_n]$. The decrease-by-one technique suggests that, the smaller-by-one problem is to generate all $(n-1)!$ permutations. 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. All the permutations obtained will be distinct and their total number will be $n(n-1)! = n!$. Hence, we will obtain all the permutations of $\{1, \dots, n\}$.

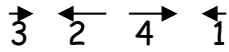
We can insert n in the previously generated permutations either left to right or right to left. Its beneficial to start with inserting n in to $1\ 2\ldots(n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1,2,\ldots,n-1\}$ needs to be processed. An example of generating permutations bottom up for $n=3$ is as follows:

Start 1; insert 2 12 21 R → L; insert 3 123 132 312 R → L
 321 231 213 L → R

The advantage of this order satisfies the minimal-change requirement; each permutation can be obtained from its immediate predecessor by exchanging just two elements in it. This algorithm is beneficial for its speed and for applications using permutations. For eg. TSP obtains various tour by the permutation process.

It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n . This is done by associating a direction in each component.

For eg.



The component K is said to be mobile if the components arrow points to a smaller number adjacent to it. For eg., here 3 and 4 are mobile but not 2 and 1. This description is called Johnson-Trotter algorithm for generating permutations.

Algorithm Johnson Trotter (n)

// Input : A positive integer n

// Output : A list of all permutations of $\{1, \dots, n\}$

← ← ←

Initialize the first permutation with 1 2 n

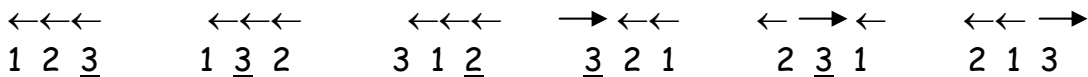
While there exists a mobile integer K do

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 .

Example: For $n=3$, with the largest mobile integer (spec.)



This algorithm is most efficient, it can be implemented to run in time proportional to the number of permutations, i.e. in $\Theta(n!)$. It is slow for large values of n .

This ordering is not natural; eg. The natural place for permutation $n \ n-1 \dots 1$ seems to be the last one on the list. This is the case if permutations are listed in increasing order, called the lexicographic order - which is the order followed similar to dictionary or index, where numbers are interpreted as alphabets:

123 132 213 231 312 321

To generate this we should follow certain things:

- If $a_{n-1} < a_n$, simply transpose these last two elements. For eg., 123 is followed by 132.
- If $a_{n-1} > a_{n-2}$ we have to engage a_{n-2}
- If $a_{n-2} < a_{n-1}$, we should rearrange the last three elements by increasing the $(n-2)$ th element as little as possible by putting there the next larger than a_{n-2} element chosen from a_{n-1} and a_n and filling positions $n-1$ and n with the remaining two of the three elements a_{n-2} , a_{n-1} and a_n in increasing order. For eg. 132 is followed by 213 while 231 is followed by 312.

In general, we scan a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$ (and hence $a_{i+1} > \dots > a_n$). Then we find the smallest digit in the tail that is larger than a_i , i.e. $\min \{a_j / a_j > a_i ; j > i\}$ and put it in position i ; the positions from $i+1$ through n are filled with the elements a_i , a_{i+1} , ..., a_n , from which the element put in the i^{th} position has been eliminated in increasing order.

• Generating Subsets:

We have seen the knapsack problem using exhaustive search approach to solve the problem by finding the most valuable subset of items that fits a knapsack of a given capacity. This is based on generating all subsets of a given set of items. Here we discuss about generating all 2^n subsets of an abstract set $A = \{a_1, a_2, \dots, a_n\}$, the set of all subsets of a set is its power set.

Using decrease-by-one concept, all subsets of $A = \{a_1, \dots, a_n\}$, can be divided into two groups; those that do not contain and those that do. The former group is ; all the subsets of $\{a_1, \dots, a_{n-1}\}$ while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \dots, a_{n-1}\}$. Thus, once we have a list of all subsets of $\{a_1, \dots, a_{n-1}\}$, we can get all the subsets of $\{a_1, \dots, a_n\}$ by adding to the list all its elements with a_n put into each of them.

An application of this algorithm to generate all subsets of $\{a_1, a_2, a_3\}$ is as below:

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

Similarly, we need not generate power sets of smaller sets for permutations. A convenient way to solve the problem is directly based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1, \dots, a_n\}$, and all 2^n bit strings b_1, \dots, b_n of length n . The easiest way for forming correspondence is to assign to a subset the bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it. For eg., the bit string 000 will correspond to a empty set of 3 element set and 110 represent $\{a_1, a_2\}$. We can generate all the bit strings of length n by generating successive binary numbers from 0 to $2^n - 1$, padded, when necessary with an appropriate number of leading 0's for eg., the case when $n=3$, is:

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

This is same as lexicographic order. Another form is squashed-order in which any subset involving a_j can be listed only after all the subsets involving a_1, a_2, \dots, a_{j-1} ($j = 1, 2, \dots, n-1$), as in three element set. It is easy to adjust the bit string-based algorithm to yield a squashed ordering of the subsets involved.

We should also find whether there exists is a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. (We need subset to differ from its immediate predecessor by either an addition or a deletion, but not both of a single element). For eg., for $n=3$, we can get,

000 001 011 010 110 111 101 100

This is an example of a Gray code. It has many properties and few useful applications.

CHAPTER 6: TRANSFORM-AND-CONQUER

INTRODUCTION:

This chapter deals with a group of design methods that are based on the idea of transformation, we call it transform-and-conquer. First, in the transformation stage, the problem's instance is modified to be, for one reason or another more relevant to solution. Then, in the second or conquering stage it is solved.

There are three major variations of this idea that differ by what we transform a given instance to :

- 1) Transformation to a simpler or more convenient instance of the same problem - call it instance simplification.
- 2) Transformation to a different representation of the same instance- representation change.
- 3) Transformation to an instance of a different problem for which an algorithm is already available - problem reduction.

Presorting is basically dealt with instance simplification. AVL trees, 2-3 trees are the combination of both instance simplification and representation change. Heaps and Heapsort are solved based on representation change. And finally, problem reduction is basically dealt with mathematical modeling or problems dealt with mathematical objects such as variables, functions and equations.

6.1 PRESORTING

Eg 1: CHECKING ELEMENT UNIQUENESS IN AN ARRAY

The Brute-Force algorithm compared pairs of the array's elements until either 2 equal elements were found or no more pairs were left. We can sort the array first and then check only its consecutive elements. If the array has equal elements, a pair of them must be next to each other and vice-versa.

```
Algm PresortElementUniqueness(A[0..n-1])
//O/p: Returns "true" if A has no equal elements, "false" otherwise
sort the array A.
for i ← 0 to n-2 do
    if A[i] = A[i+1] return false
return true
```

Time is based on sorting and checking the uniqueness. Sorting, it takes $n \log n$ comparisons and for checking $n-1$ comparison. Therefore, the worst-case efficiency is $\theta(n \log n)$.

$$\begin{aligned} T(n) &= T_{\text{sort}}(n) + T_{\text{scan}}(n) \\ &= \theta(n \log n) + \theta(n) \\ &= \theta(n \log n). \end{aligned}$$

Eg 2: COMPUTING A MODE

A mode is a value that occurs most often in a given list of numbers. The Brute-Force method is to scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. Enter the values with their frequencies in a separate list. On each iteration, the i^{th} element of the original list is compared with the values already encountered by traversing this auxiliary list. If matching is found, its frequency is incremented otherwise, the current element is added to the list of distinct values seen so far with the frequency of 1.

The worst-case comparisons is:

$$c(n) = \sum_{i=1}^n (i-1) = 0+1+\dots+(n-1) = \frac{n(n-1)}{2} \in \theta(n^2)$$

Algm PreSortMode ($A[0..n-1]$)

//O/p: The array's mode

sort the array A

$i \leftarrow 0$

modefrequency $\leftarrow 0$

while $i \leq n-1$ do

 runlength $\leftarrow 1$; runvalue $\leftarrow A[i]$

 while $i + \text{runlength} \leq n-1$ and $A[i + \text{runlength}] = \text{runvalue}$

 runlength $\leftarrow \text{runlength} + 1$

 if runlength > modefrequency

 modefrequency $\leftarrow \text{runlength}$;

 modevalue $\leftarrow \text{runvalue}$

$i \leftarrow i + \text{runlength}$

return modevalue

The running time will be linear, whose worst-case will be $n \log n$ based on the sorting.

Eg 3: SEARCHING PROBLEM

Consider the problem of searching for a given value v in a given array of n sortable items. The Brute-Force solution is sequential search that needs n comparisons in the worst case. If the array is sorted first, we can then apply binary search that requires only $(\lceil \log_2 n \rceil + 1)$ comparisons in the worst case. Assume, that the most efficient $n \log n$ sort, the total running time of such a searching algorithm in the worst case will be

$$\begin{aligned} T(n) &= T_{\text{sort}}(n) + T_{\text{search}}(n) \\ &= \theta(n \log n) + \theta(\log n) \\ &= \theta(n \log n) \end{aligned}$$

6.2 Balanced Search Trees:

The binary search tree-one of the principal data structures for implementing dictionaries. It is binary trees whose nodes contains a set of orderable items, one element per node, so that all elements in the left sub tree are smaller than the element in the sub tree's root and all the elements in the right sub tree are greater than it. The transformation is an example of the representation change technique. The operations searching, insertion and deletion are in $\theta(\log n)$ in the average case. The worst case is in $\theta(n)$ because the tree can degenerate into a severely unbalanced one with height equal to $n-1$.

There are two approaches:

- The first approach is of the instance simplification variety i.e., an unbalanced binary search tree is transformed to a balanced one. An AVL tree requires the difference between the heights of the left and the right sub trees of every node never exceed 1. A red-black tree tolerates the height of one subtree being twice as large as the other subtree of the same node. If the tree is unbalanced do the rotation and make it balanced one. Information about other types of binary search trees that utilize the idea of rebalancing via rotations, including red-black trees and so-called splay trees.
- The second approach is of representation change: it allows more than one element in a node of a search tree. Such trees are 2-3, 2-3-4 and B-trees. They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced.

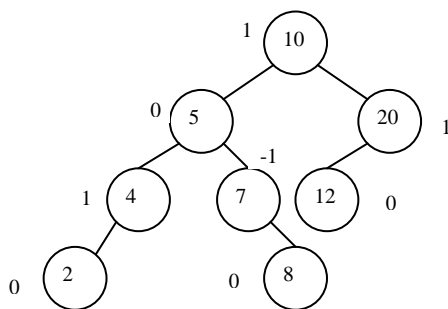
AVL Trees:

AVL trees - 1962 - Russian scientists - Adelson and Landis. AVL tree is defined as a binary search tree in which the balance factor of every node, which is defined as the difference between the heights of the node's left and right sub trees, is either 0 or +1 or -1. The height of the empty tree is defined as -1.

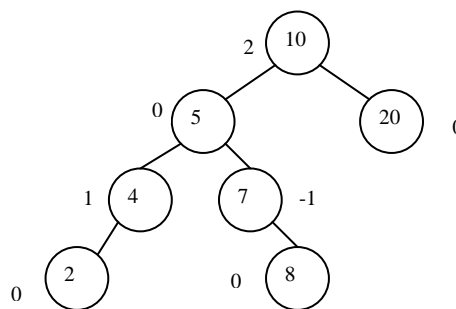
If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A rotation in an AVL is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2; if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf. Four rotations are there, which is of simple form Left and Right rotations and double form i.e., LR and RL rotations.

The single right rotation or R-rotation in its most general form represents the rotation is performed after a new key is inserted into the left sub tree of the left child of a tree whose root had the balance of +1 before the insertion.

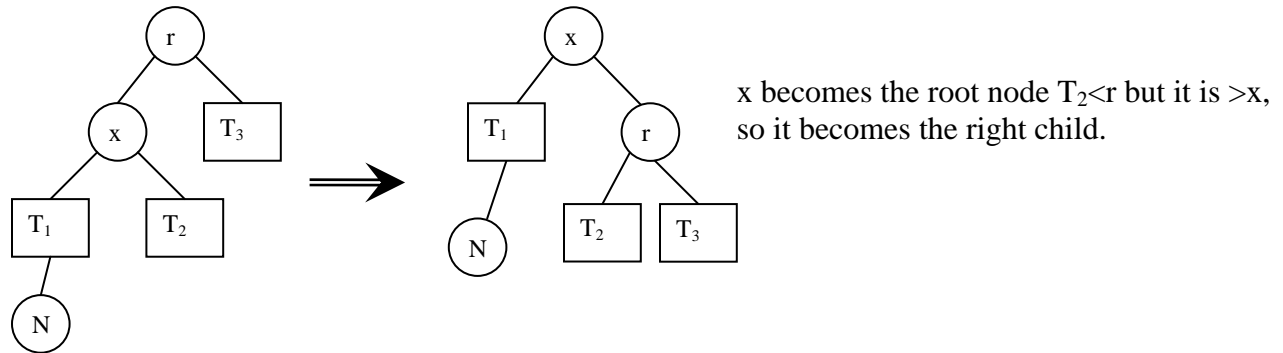
The single left rotation or L-rotation is the mirror image in which, it is performed after a new key is inserted into the right sub tree of the right child of a tree whose root had the balance of -1 before the insertion.



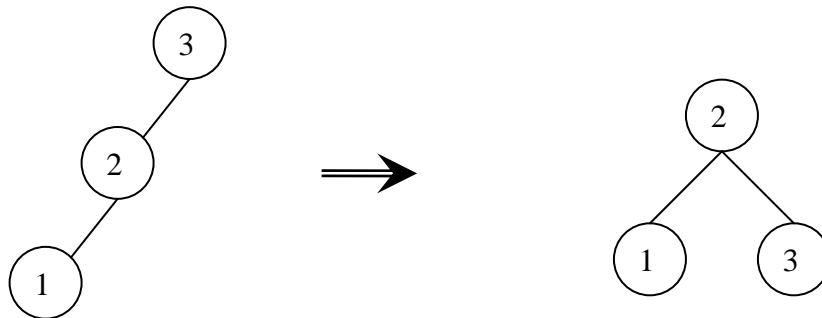
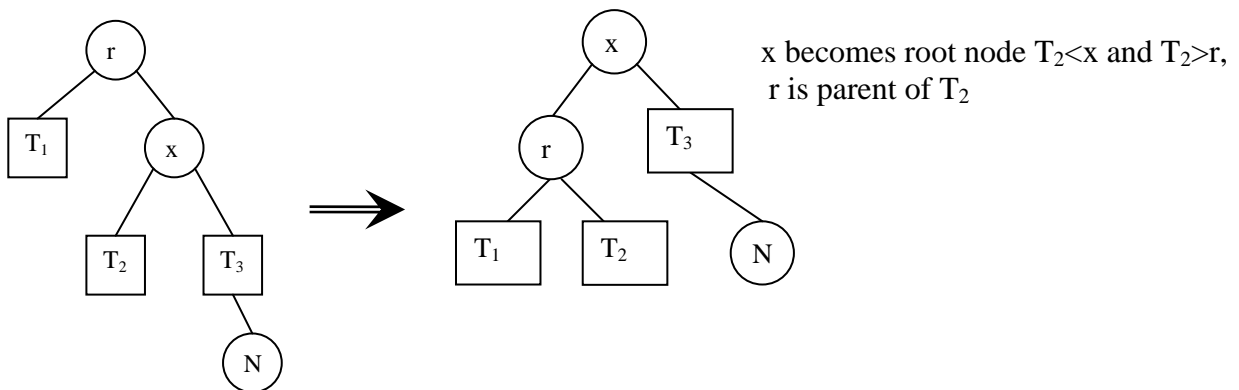
This is a AVL tree



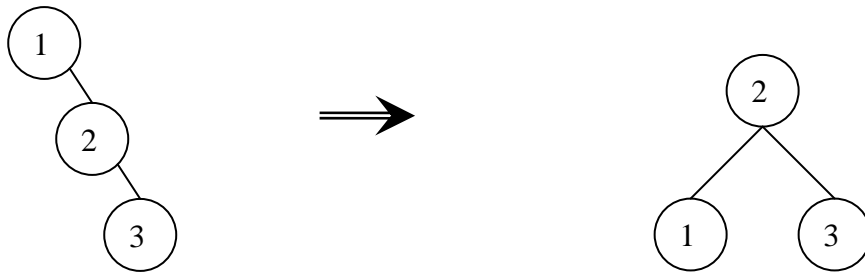
This is not a AVL tree

Right Rotations (Left high)

Eg :-

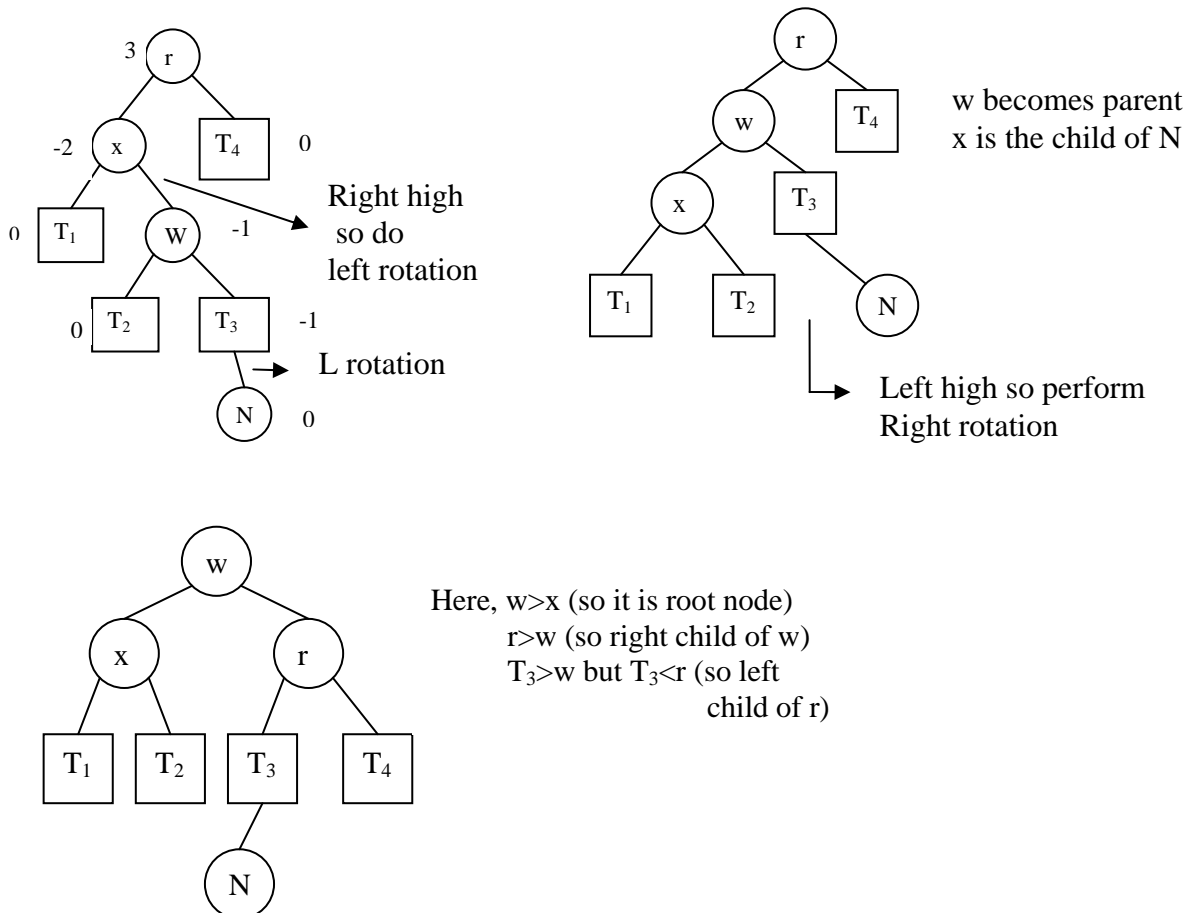
**Left rotation (Right high):**

Eg:-

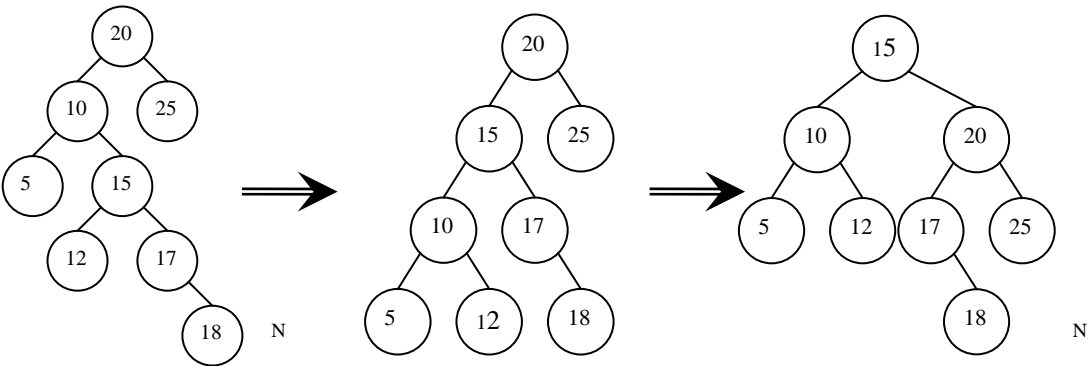


The second notation type is called double left-right notation (LR), it is a combination of the above two: We perform the L-rotation of the left sub tree of root r followed by the R-rotation of the new tree rooted at r . It is performed after a new key is inserted into the right sub tree of the left child of a tree whose root had the balance of +1 before the insertion.

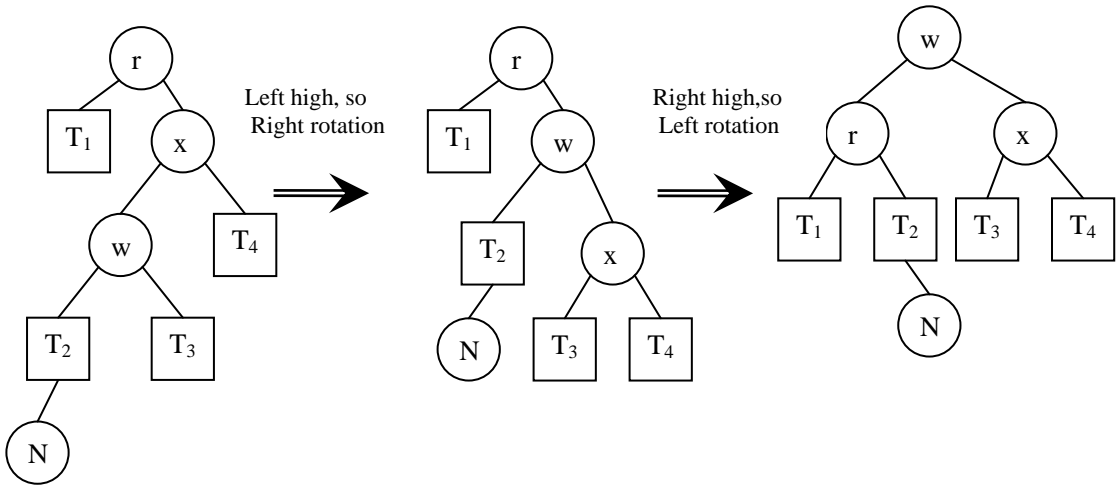
LR(Left rotation then right rotation)



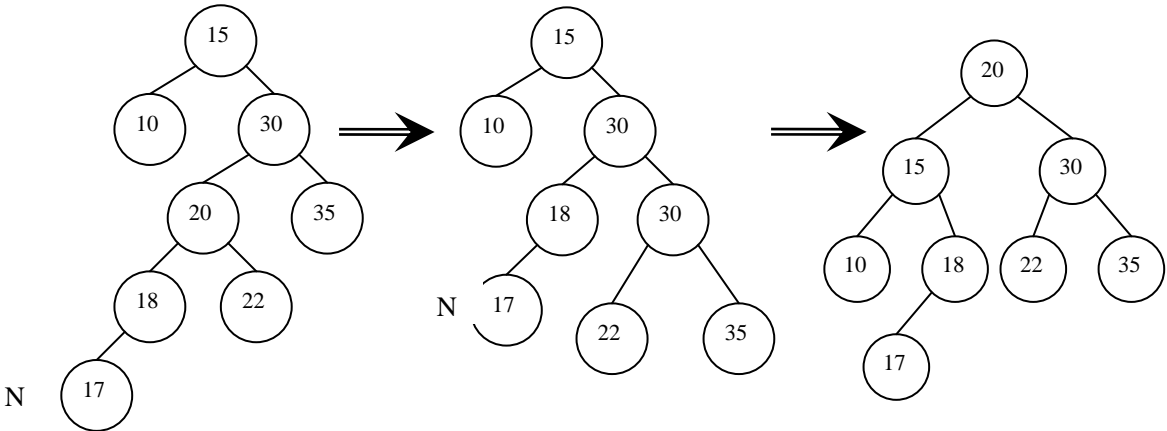
Eg.



RL-Rotation



E.g:



Note : Left sub tree of right child (-1) before insertion. We perform R-rotation of right sub tree of root r followed by L-rotation of new tree rooted at r.

The characteristics are based on the tree's height. It turns out that it is bounded both above and below by logarithmic function. The height h of any AVL tree with n nodes.

$$\log_2 n \leq h < 1.4405 \log_2(n+2) - 1.3277$$

The operations of searching and insertion are $\Theta(\log n)$ in the worst case. Searching in an AVL tree is difficult than insertion. The drawbacks of these are frequent rotations, the need to maintain balances for the tree's nodes, especially of the deletion operation.

2-3 Trees:

A 2-3 is a tree that can have nodes of 2 kinds: 2 nodes and 3 nodes.

- A 2-node contains a single key k and has two children: The left child serves as the root of a sub tree whose keys are less than k and the right child serves as a root of a sub tree whose keys are greater than k.
- A 3-node contains two ordered keys k1 and k2 ($k_1 < k_2$) and has 3 children. The left most child serves as the root of a sub tree with keys less than k1, the middle child serves as the root of a sub tree with keys between k1 and k2, and the right most child serves as the root of a sub tree with keys greater than k2.

The last requirement is that all its leaves must be on the same level, i.e., a-2-3-tree is always height balance: The length of a path from the root of the tree to a leaf must be the same for every leaf.

For searching, compare with the root node, if it is equal to the root, stop or continue left and right based on smaller and larger than the root key respectively.

For insertion, insert a new key k at a leaf except for the empty tree. If it is a 2-node, insert as first or second key depending on whether k is smaller or larger than the node's old key. If the leaf is a 3-node, we split the leaf in two: The smallest of the 3 keys is put in the first leaf, largest in the second leaf, middle key is promoted to the old leaf's parent. Avoid overflow during insertion by splitting it further.

The efficiency is determined based on tree's height. For any 2-3 tree of height h with n -nodes, the inequality is

$$n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1 \text{ and hence,}$$

$$h \leq \log_2(n+1) - 1$$

On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and 3 children. Therefore, for any 2-3 trees with n nodes,

$$n \leq 2.1 + 2.3 + \dots + 2.3^h = 2(1 + 3 + \dots + 3^h) = 3^{h+1} - 1$$

and hence, $h \geq \log_3(n+1) - 1$

These lower and upper bounds on height h ,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

The time efficiencies are thus $\Theta(\log n)$ in both worst case and average case in searching, insertion and deletion.

6.3 Heaps and heap sort

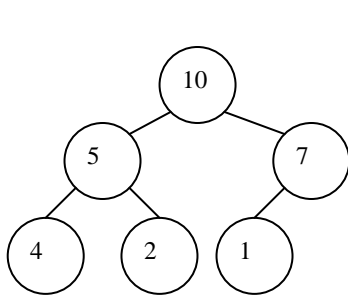
The data structure called the heap ordered pile of items. A priority queue is a set of items with an orderable characteristic called an item's priority with the operation:

- ♠ Finding an item with the highest priority.
- ♠ Deleting an item with the highest priority.
- ♠ Adding a new item to the set.

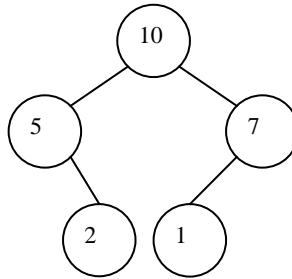
Definition: A heap can be defined as a binary tree with keys assigned to its nodes with the following conditions:

1. The tree's shape requirement - The binary tree is essentially complete, i.e., all its levels are full except possibly the last level, where only some right most leaves may be missing.
2. The parental dominance requirement - The key at each node is greater than or equal to the keys at its children.

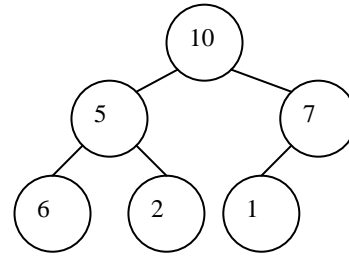
E.g.



This is a heap



Not a heap, the tree's shape requirement is violated



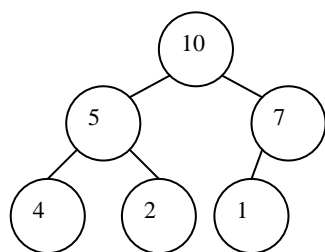
Not a heap, the parental dominance requirement is not satisfied

The properties of the heap are:

1. There exists one essentially complete binary tree with n nodes. Its height is equal to $\log_2 n$.
2. The root of a heap always contains its largest element.
3. A root of a heap considered with all its descendents is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's element in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. Here,
 - a. The parental node keys will be in the first $n/2$ positions.
 - b. The children of a key in the array's parental position i ($1 \leq i \leq n/2$) will be in positions $2i$ and $2i + 1$ and the parent of a key in position i ($2 \leq i \leq n$) will be in position $[i/2]$.

Heap can also be defined as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$ i.e.,

$$H[i] \geq \max \{ H[2i], H[2i + 1] \} \text{ for } i = 1, 2, \dots, n/2.$$



index	value	0	1	2	3	4	5	6
			10	5	7	4	2	1

There are two principal alternatives for constructing a heap for a given list of keys. The first is bottom-up heap construction. It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then heapifies the tree as follows. Starting with the last parental node and ending with the root, the algorithm checks whether the parental dominance holds for the key at this node. If it does not, the algorithm exchanges the node's key k with the larger key of its children and checks whether the parental dominance holds for k in its new position. This continues until the parental dominance is satisfied.

Algorithm HeapBottomUp($H[1..n]$)

// Constructs a heap by bottom-up
 // Input: An array $H[1..n]$ of ordered I items
 // Output: A Heap $H[1..n]$

```

for  $i \leftarrow n/2$  down to 1 do
   $k \leftarrow i$ ;  $v \leftarrow H[k]$ 
  heap  $\leftarrow$  false
  while not heap and  $2*k \leq n$  do
     $j \leftarrow 2*k$ 
    if  $j < n$  // there are two children
      if  $H[j] < H[j+1]$ 
         $j \leftarrow j+1$ 
      if  $v \geq H[j]$ 
        heap  $\leftarrow$  true
      else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$ 
   $H[k] \leftarrow v$ 
  
```

Analysis:

Efficiency in the worst case: Assume $n = 2^h - 1$, so that a heap tree is full, i.e., the largest number of nodes occur on each level. 'h' is the height of the tree, i.e., $h =$

$\log_2 n$ The total number of tree comparisons on level i is $2(h-i)$. So, the total number of key comparisons in the worst case is :

$$C_{\text{worst}}(n) = \sum_{i=0}^{h-1} \sum_{\substack{\text{level } i \\ \text{keys}}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i) 2^i = 2(n - \log_2(n+1))$$

The second alternative is top-down heap construction algorithm. First, attach a new node with key k in it after the last leaf of the existing heap. Then shift k up to its appropriate place in the new heap as follows: Compare k with its parent's key: If the latter is greater than or equal to k , stop, otherwise, swap these two keys and compare k with its new parent. It continues until k is not greater than its last parent or it reaches the root.

The algorithm is given for deleting a node.

Step 1: Exchange the root's key with the last key k of the heap.

Step 2: Decrease the heap's size by 1.

Step 3: Heapify the smaller tree by sifting k down the tree exactly in the same way we did it in the bottom-up, i.e., verify the parental dominance for k : If it holds, we are done; if not, swap k with the larger of its children.

Heap sort:

This is a 2-stage algorithm that works as follows:

Stage 1 (Heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root deletion operation $n-1$ times to the remaining heap.

$$\begin{aligned} C(n) &\leq 2[\log_2(n-1)] + 2[\log_2(n-2)] + \dots + 2[\log_2 1] \\ &\leq 2 \sum_{i=1}^n \log_2 i \leq 2 \sum_{i=1}^{n-1} \log_2(n-1) \\ &= 2(n-1) \log_2(n-1) \leq 2n \log_2 n \end{aligned}$$

This means that $c(n) \in O(n \log n)$ for second stage.

For both stages, $O(n) + O(n \log n) = O(n \log n)$.

Stage 1

```

2 9 7 6 5 8
2 9 8 6 5 7
2 9 8 6 5 7
9 2 8 6 5 7
9 6 8 2 5 7

```

Stage 2

```

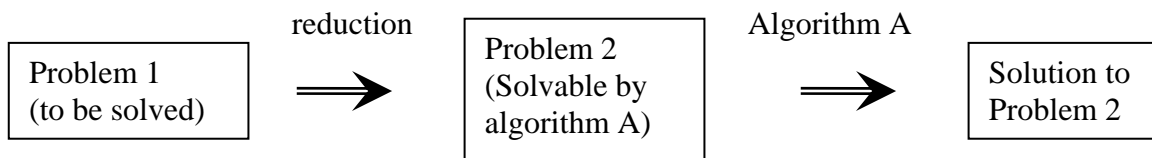
[9] 6 8 2 5 7
7 6 8 2 5 | [9]
[8] 6 7 2 5
5 6 7 2 | [8]
[7] 6 5 2
2 6 5 | [7]
  [6] 2 5
    5 2 | [6]
      [5] 2
        2 | [5]
          2

```

The final sorted array is: 2 5 6 7 8 9

6.4 Problem reduction:

The problem will be reduced to another problem that you know how to solve.

**Computing the LCM:**

For finding the LCM of 2 numbers, first approach is to find the prime factors and solve. The second method is to find the GCD of 2 numbers and solve.

$$\text{lcm}(m, n) = m.n/\text{gcd}(m,n)$$

Counting paths in a graph:

The problem can be solved with an algorithm for computing an appropriate power of its adjacency matrix.

Reduction of optimization problems:

If a problem seeks to find a maximum of some function, it is said to be maximization problem

$$\max f(x) = -\min[-f(x)].$$

$$\min f(x) = -\max[-f(x)]$$

Maximization can be solved by minimization and vice versa.

Linear programming:

There are different problems like simplex method, continuous or fractional version, discrete or 0-1 version eg: Knapsack problem.

Reduction to graph problems:

In these applications, vertices of a graph typically represent possible states of the problem in question while edges indicate permitted transition among such states. One of the graph's vertices represents an initial state, while another represents a goal state of the problem. Such a graph is called a state-state graph.

Chapter 7. Space and time tradeoffs

Introduction:

It is a well-known issue for both theoreticians and practitioners. An example, is the problem of computing values of a function at many points in its domain. We can precompute the function's value and store in a table. In general, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterwards. It is called an input enhancement. The algorithms followed based on these are as follows:

1. Counting methods for storing.
2. Boyer-Moore algorithm for string matching and its simplified version-Horspool's algorithm.

The other technique that exploits space and time tradeoffs uses extra space to facilitate faster and/or more flexible access to the data called prestructuring. Some processing are done before a problem in question is actually solved, for this, there are 2 approaches:

1. Hashing.
2. Indexing with B-trees.

There is one more algorithm based on this, called dynamic programming. It is based on recording solutions to overlapping sub problems of a given problem in a table from which a solution to the problem in question is then obtained.

Comments:

- The two resources, time and space, do not have to compete with each other in all design situations. Minimize both running time and space consumed for an efficient algorithm. Use a space-efficient data structure to represent a problem's input which leads, in turn to a faster algorithm. E.g., Traversing the graph based on BFS and DFS. The adjacency matrix [$\Theta(n^2)$] representation and linked list [$\Theta(n + m)$] representation decides the time efficiency.
- Data compression is most important for space and time tradeoffs. Size reduction is goal in solving the problem efficiently.

7.1 Sorting by counting:

For each element for a list to be sorted, the total number of elements smaller than this element and record the results in a table. The numbers indicate the position of the elements in the sorted list. Eg., if count is 10, the position is 11 (i.e., index starts with 0). Copy the elements to their appropriate positions from the table to make a new sorted list, called comparison counting.

Array A[0...5]

62	31	84	96	19	47
----	----	----	----	----	----

Initially count[]

After pass i=0 count[]

.

.

.

After pass i=4 count[]

Final state count[]

0	0	0	0	0	0
3	0	1	1	0	0
.					
.					
.					
				0	2
3	1	4	5	0	2

Array S[0...5]

19	31	47	62	84	96
----	----	----	----	----	----

Algorithm comparison counting sort (A[0...n-1])

// Sorts an Array

// Input: A[0..n-1] of orderable values

// Output: S[0..n-1] of A's elements sorted in increasing order

```

for i ← 0 to n-1 do count[i] ← 0
    for i ← 0 to n-2 do
        for j ← i+1 to n-1 do
            if A[i] < A[j]
                count[j] ← count[j] + 1
            else
                count[i] ← count[i] + 1
for i ← 0 to n-1 do S[count[i]] ← A[i]
return s.
```


Analysis:

The number of times the basic operation ($A[i] < A[j]$) executed is :

$$\begin{aligned}
 c(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] \\
 &= \sum_{i=0}^{n-2} (n-1-i) = [n(n-1)]/2 \in \theta(n^2)
 \end{aligned}$$

Counting works efficiently in a set of small values. If 1 and 2 is there, check the number of times of its occurrences. If element values are integers between lower bound l and upper bound u , the frequency is computed as $F[0..u-l]$, the first $F[0]$ positions must be filled with l , the next $F[1]$ positions with $l+1$ etc.,

The elements of A whose values are less are copied from positions 0 through $F[0]-1$, the elements of value $l+1$ are copied from $F[0]$ to $(F[0] + F[1]) - 1$, etc. Such sums of frequencies are called distribution counting.

E.g: $A[0..5]$

13	11	12	13	12	12
----	----	----	----	----	----

the set is {11,12,13}. The frequency and distribution are:

Array Values	11	12	13
Frequency	1	3	2
Distribution	1	4	6

	D[0..2]			S[0..5]		
A[5]=12	1	4	6			12
A[4]=12	1	3	6		12	
A[3]=13	1	2	6			13
A[2]=12	1	2	5		12	
A[1]=11	1	1	5	11		
A[0]=13	0	1	5			13

Each time decrease

distribution by 1.

Algorithm DistributionCounting (A[0..n-1])

```

// Sorts an array
// Input: A[0..n-1] of integers between l & u (l ≤ u)
// Output: S[0..n-1] of A's elements sorted in increasing order

for j ← 0 to u-l do D[j] ← 0           // initialize frequencies
for i ← 0 to n-1 do D[A[i]-l] ← D[A[i]-l]+1 // compute frequencies
for j ← 1 to u-l do D[j] ← D[j-1]+D[j]   // reuse for distribution
for i ← n-1 down to 0 do
    j ← A[i]-l
    S[D[j]-1] ← A[i]
    D[j] ← D[j]-1
return S

```

It is a better time efficiency than merge sort, quick sort, etc, which gives a linear value.

7.2 Input enhancement in string matching:

The problem is to find an occurrence of a given string of 'm' characters (pattern) in a string of 'n' characters (text). In Brute-Force method, it scans the text from left to right and if a mismatch occurs, shifts by 1 position to the right. The number of trials is $n-m+1$ and number of comparisons is m .

The worst case comparison is $m(n-m+1) \in \Theta(nm)$. It is good for practical applications, but bad since there can be a faster algorithm.

The input enhancement idea is to preprocess the pattern to get some information and store it in a table, and then use the information during the search for the pattern in the text. There are 2 algorithms for this: Knuth Morris-Pratt and Boyer-Moore algorithm.

The difference between the two is: In Knuth Morris-Pratt, the scan is from left to right, while in Boyer-Moore, it is left to right starting from the last character in the pattern. Boyer-Moore is simple, but still is less frequently used. The simplified version of this is Horspool's algorithm which is widely used.

Horspool's algorithm:


Consider the example as BARBER pattern in a text

$S_0 \dots C \dots S_{n-1}$
BARBER


Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. If there is a mismatch then shift the pattern towards the right, to shift we should know the shift size by looking at character *C* of the text that was aligned against the last character, in the pattern.

In general, the following four possibilities can occur:


Case 1: If there are no *C*'s in the pattern, shift the pattern by its entire length to right.

e.g : $S_0 \dots S \dots S_{n-1}$

BARBER
BARBER

Case 2: If there are occurrences of character '*c*' in the pattern but the last one, then shift should align the right most occurrence of *c* in the pattern

$S_0 \dots B \dots S_{n-1}$

BARBER
BARBER

Case 3: If *C* happens to be the last character in the pattern then there are no *C*'s among its *m*-1 character, shift same as case 1.

$S_0 \dots M \ E \ R \dots S_{n-1}$

LEADER
LEADER

Case 4: If C happens to be the last character in the pattern and there are other C 's among its first $n-1$ characters the shift is same as case 2.

$S_0 \dots \dots \dots O \ R \dots \dots \dots S_{n-1}$


 REORD E R
 RE O R DER

Input enhancement makes repetitive comparisons unnecessary. Shift sizes are precomputed and stored in a table. The shift value is calculated by the formula:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the } 1^{\text{st}} m-1 \text{ characters of} \\ \text{the pattern to its last character, otherwise} \end{cases}$$

For eg.: The table entries will be equal to 6, except for E, B, R and A which will be 1, 2, 3 and 4 respectively.

The algorithm works as:

- First initialize all entries of m .
- Scan left to right $m-1$ times.
- For the j^{th} character of the pattern ($0 \leq j \leq m-2$) overwrite its entry in the table with $m-1-j$, which is the character's distance to the right end of the pattern.

Algorithm Shifttable($p[0..m-1]$)

// Fills the table by Horspool's & Boya-Moore

// Input: pattern $p[0..m-1]$ and an alphabet of possible characters

// Output: Table[$0..size-1$] indexed by the alphabet's characters and filled with shift sizes computed using $t(c)$

initialize all the elements of Table with m .

for $j \leftarrow 0$ to $n-1$ do Table[$p[j]$] $\leftarrow m-1-j$.

return table.

Now the algorithm is summarized as follows:

Horspool's Algorithm:

- Step 1:** For a given pattern of length m and the alphabet used in both the pattern and text construct a shift table.
- Step 2:** Align the pattern against the beginning of the text.
- Step 3:** Repeat the following until either a matching sub string is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched or a mismatching pair is encountered. If mismatch, retrieve the entry $t(c)$ from the c 's column of the shift table where c is the text character currently aligned against the last character of the pattern and shift the pattern by $t(c)$ characters to the right along the text.

Algorithm HorspoolMatching($P[0..m-1]$, $T[0..n-1]$)

// Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$

// Output: The index of the left end of the first matching substring or -1 if there are no matches

shift table($P[0..m-1]$) // generates table of shifts

$i \leftarrow m-1$ // position of the pattern's right end

while $i \leq n-1$ do

$k \leftarrow 0$ // number of matched characters

 while $i \leq m-1$ and $P[m-1-k] = T[i-k]$

$k \leftarrow k+1$

 if $k = m$

 return $i-m+1$

 else $i \leftarrow i + \text{Table}[T[i]]$

return -1.

Eg: Pattern-BARBER

Shift table :

Character c	A	B	C	D	E	F	R	Z	-
Shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Actually search proceeds as follows:

JIM-SAW-ME-IN-A-BARBER SHOP

```

BARBER  BARBER
  BARBER BARBER
    BARBER  BARBER
  
```

The worst-case efficiency is $\Theta(nm)$. But for random texts, it is in $\Theta(n)$. Horspool's is obviously faster on average than Brute-Force algorithm.

Boyer-Moore Algorithm:

Boyer-Moore algorithm is similar to Horspool's, but it act differently, after some positive number k ($0 < k < m$) of the pattern's characters are matched successfully before a mismatch is encountered.

```

S0.....C.....Si-k+1.....Si.....Sn-1 text
      //      ||      ||
P0...Pm-k-1 Pm-k      Pm-1
  
```

It determines the shift size by considering the 2 quantities. The first one is guided by the text's character c that caused a mismatch, called bad-symbol shift. If c is not the pattern, we shift the pattern to just pass this ' c ' in the text. Shift is computed as $t_1(c) - k$, where $t_1(c)$ is the entry in the precomputed table and k is the number of matched characters.

$$d_1 = \max\{t_1(c) - k, 1\}$$

If $t_1(c) - k > 0$ used when the mismatching character c of the text occurs in the pattern. If $t_1(c) - k \leq 0$ we need not shift the pattern. So we call Brute-Force for moving one character or position of the pattern to the right.

The second type is based on $k > 0$ characters in the pattern. The ending portion of the pattern as its suffix of size k and denote as $\text{suff}(k)$ good-suffix shift.

Boyer-Moore algorithm:

Step 1: For a given pattern and the alphabet used is both the pattern and the text, construct the bad-symbol shift as described.

Step 2: Using the pattern, construct the good-suffix shift table as described.

Step 3: Align the pattern against the beginning of the text.

Step 4: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the characters in the pattern and text until either all m characters are matched or a mismatching pair is encountered after $k \geq 0$ character pairs are matched.

$$d = \begin{cases} d_1 & \text{if } k=0 \\ \max\{d_1, d_2\} & \text{if } k>0 \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$

Note: Boyer-Moore algorithm is also in a linear form.

7.3 Hashing:

It is based on the idea of distributing keys among a one-dimensional array $H[0..m-1]$ called a hash table. The distribution is done by computing, for each of the keys, predefined function h called the hash function. This function assigns an integer between 0 & $m-1$, called hash address to key. For eg, if keys are non negative integers, a hash function can be of the form $h(k) = k \bmod m$. If keys are letters of some alphabet, we can first assign a letter to its position in the alphabet and then apply the same kind of a function used for integers. Finally, if k is a character string c_0, c_1, \dots, c_{s-1} ,

we use $(\sum_{i=0}^{s-1} \text{ord}(c_i)) \bmod m$; a better option to compare $h(k)$ is:

$h \leftarrow 0$; for $i \leftarrow 0$ to $s-1$ do $h \leftarrow (h * c + \text{ord}(c_i)) \bmod m$ where c is a constant greater than every $\text{ord}(c_i)$.

The 2 requirements are:

- a. A hash function needs to distribute keys among the cells of the hash table as evenly as possible.
- b. A hash function has to be easy to compute.

Collision occurs when we choose a hash table's size m to be smaller than the number of keys ' n ' - a phenomenon of two or more keys being hashed into the same cell of the hash table. The resolution mechanism for collision is of 2 types open (Separate chains) & closed hashing (open addressing).

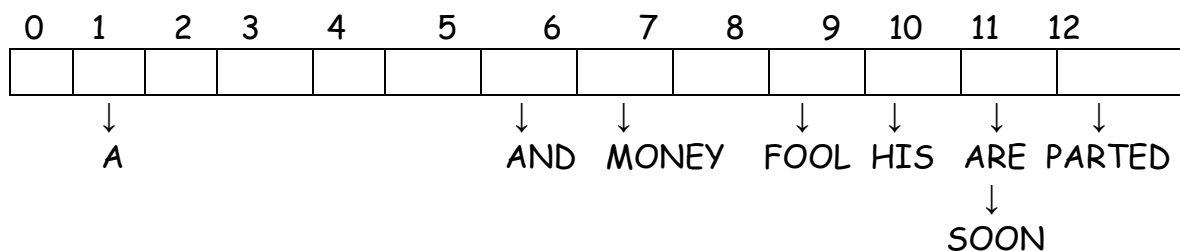
Open Hashing:

It is stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell.

E.g. A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED.

$h(A)=1 \bmod 13=1$ (ie., 1st cell). Fool - in 9th cell (since $(6+15+15+12) \bmod 13=9$) etc. ARE & SOON=11th cell.

Keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
Hash address	1	9	6	10	7	11	11	12



Note : If KID has to be entered it is entered in the 11th cell
i.e., $KID (11 + 9 + 4) \bmod 13 = 11$

If the hash function distribute n keys among m cells of the hash table about evenly, each list is n/m keys long Ratio $\alpha=n/m$ called load factor. The pointers or chain links inspected for successful and unsuccessful search.

$$S \approx 1 + \alpha / 2 \quad \& \quad U = \alpha$$

Two dictionary operations- insertion and deletion are almost identical to searching. For average case, the efficiency is $\Theta(1)$, if the number of keys is n and table size is m .

Closed Hashing:

The simplest resolution - called linear probing checks the cell following the one where the collision occurs. If that cell is empty, the new key is installed there; if the next cell is already occupied, the availability of that cell's immediate successor is checked, and so on. If the end of the table is reached, the search is wrapped to the beginning of the table; that is, it is treated as circular array.

$$S \approx \frac{1}{2}(1+(1/(1-\alpha))) \quad \& \quad U \approx \frac{1}{2}(1+(1/(1-\alpha)^2))$$

A cluster in linear probing is a contiguously occupied cells. It makes the operation less efficient. Double hashing is another form, in which we use another hash function, $s(k)$, to determine a fixed increment for the probing sequence to be used after a collision at location $l=h(k)$:

$$(l+s(k)) \bmod m, (l+2s(k)) \bmod m, \dots$$

Rehashing is a situation where the current table is scanned, and all its keys are relocated into a larger table.

For storing a symbol table, its useful for storing very large dictionaries on disks-called extendible hashing. A location computed by the hash function in extendible hashing indicates a disk address of a bucket that can hold up to b keys. When a key's bucket is identified, all the bucket's keys are lead into main memory and then scanned for the key in question.

Chapter 8. Dynamic Programming

Introduction:

Invented in 1950 by Richard Bellman an U.S mathematician as a general method for optimizing multistage decision processes. It is a planning concept, which is a technique for solving problems with overlapping subproblems. This arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping subproblems it suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem. It can also be used for avoiding using extra space.

Bottom-up approach all the smaller subproblems to be solved. In top-down it avoids solving the unnecessary subproblems, which is called memory-functions.

8.1 Computing a Binomial Coefficient

Its used to apply in nonoptimization problem. Elementary combinatorics we have, binomial coefficient, denoted $c(n,k)$, is the number of combinations k elements from an n -element set ($0 \leq k \leq n$).The binomial formula is:

$$(a+b)^n = c(n,0)a^n + \dots + c(n,i)a^{n-i}b^i + \dots + c(n,n)b^n$$

$$c(n,k) = c(n-1,k-1) + c(n-1,k) \text{ for } n > k > 0 \text{ and } c(n,0) = c(n,n) = 1$$

	0	1	2	k-1	k
0	1					
1	1	1				
2	1	2	1			
⋮						
⋮						
⋮						
k	1					1
⋮						
⋮						
⋮						
n-1	1				c(n-1,k-1)	c(n-1,k)
n	1					c(n,k)

Algorithm Binomial (n,k)

// Computes c(n,k)

// Input: A pair of non -ive int $n \geq k \geq 0$

// Output: The value of c(n,k)

for $i \leftarrow 0$ to n do

for $j \leftarrow 0$ to $\min(i,k)$ do

if $j=0$ or $j=k$

$c[i,j] \leftarrow 1$

else $c[i,j] \leftarrow c[i-1, j-1] + c[i-1, j]$

return $c[n,k]$

The time efficiency is based on the basic operation i.e., addition. Also the 1st $k+1$ rows of the table form a triangle, while the remaining $n-k$ rows form a rectangle, we have to split the sum expressing $A(n,k)$ into 2 parts

$$\begin{aligned}
 A(n,k) &= \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^n \sum_{j=1}^k 1 = \sum_{i=1}^k (i-1) + \sum_{i=k+1}^n k \\
 &= [(k-1)k]/2 + k(n-k) \in \Theta(nk)
 \end{aligned}$$

8.2 Warshall's and Floyd's Algorithms

Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all pairs shortest-paths problem.

Warshall's algorithm:

The adjacency matrix $A=\{a_{ij}\}$ of a directed graph is the Boolean matrix that has 1 in its i^{th} row and j^{th} column iff there is a directed edge from the i^{th} vertex to j^{th} vertex.

Definition:- The transitive closure of a directed graph with n vertices can be defined as the $n \times n$ Boolean matrix $T=\{t_{ij}\}$, in which the element in the i^{th} row ($1 \leq i \leq n$) and the j^{th} column ($1 \leq j \leq n$) is 1 if there exists a nontrivial directed path (i.e., directed path of positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.

We can generate it with DFS or BFS .It constructs the transitive closure of a given digraph with n vertices through a series of $n \times n$ matrices.

$$R^0, R^{(1)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $rij^{(k)}$ in the i^{th} row and j^{th} col of matrix $R^{(k)} = 1$ iff there exists a directed path from the i^{th} vertex to j^{th} vertex with each intermediate vertex, if any, not numbered higher than k . It starts with $R^{(0)}$ which doesnot allow any intermediate vertices in its paths; $R^{(1)}$ use the 1^{st} vertex as intermediate; i.e., it may contain more ones than $R^{(0)}$. Thus, $R^{(n)}$ reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the digraph's transitive closure.

$R^{(k)}$ is computed from its immediate predecessor $R^{(k-1)}$. Let $rij^{(k)}$, the element in the i^{th} row and j^{th} col of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from i^{th} vertex v_i to the j^{th} vertex v_j with each intermediate vertex numbered not higher than k .

v_i , a list of intermediate vertices each numbered not higher than k, v_j .

There are 2 possibilities. First is, if there is a list the intermediate vertex doesnot contain the k^{th} vertex. The path from v_i to v_j has vertices not higher than $k-1$, and therefore $rij^{(k-1)}=1$. The second is, that path does contain the k^{th} vertex v_k among the intermediate vertices. v_k occurs only once in the list.

Therefore v_i , vertices numbered $\leq k-1$, v_k , vertices numbered $\leq k-1$, v_j .

Means there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ hence $r_{ik}^{(k-1)} = 1$ and the 2nd part is such that a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ hence, $r_{kj}^{(k-1)} = 1$ i.e., if $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$.

Thus the formula is : $r_{ij}^{(k)} = r_{ij}^{(k-1)}$ or $r_{ik}^{(k-1)}$ and $r_{kj}^{(k-1)}$

This formula yields the Warshall's algorithm which implies:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an elt r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ iff the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

$$R^{(k-1)} = \begin{matrix} & \begin{matrix} j & k \end{matrix} \\ \begin{matrix} k \\ i \end{matrix} & \begin{bmatrix} 1 & & \\ 0 & & 1 \end{bmatrix} \end{matrix} \Rightarrow R^{(k)} = \begin{matrix} & \begin{matrix} j & k \end{matrix} \\ \begin{matrix} k \\ i \end{matrix} & \begin{bmatrix} 1 & & \\ 1 & & 1 \end{bmatrix} \end{matrix}$$

Algorithm Warshall ($A[1..n,1..n]$)

$R^{(0)} \leftarrow A$

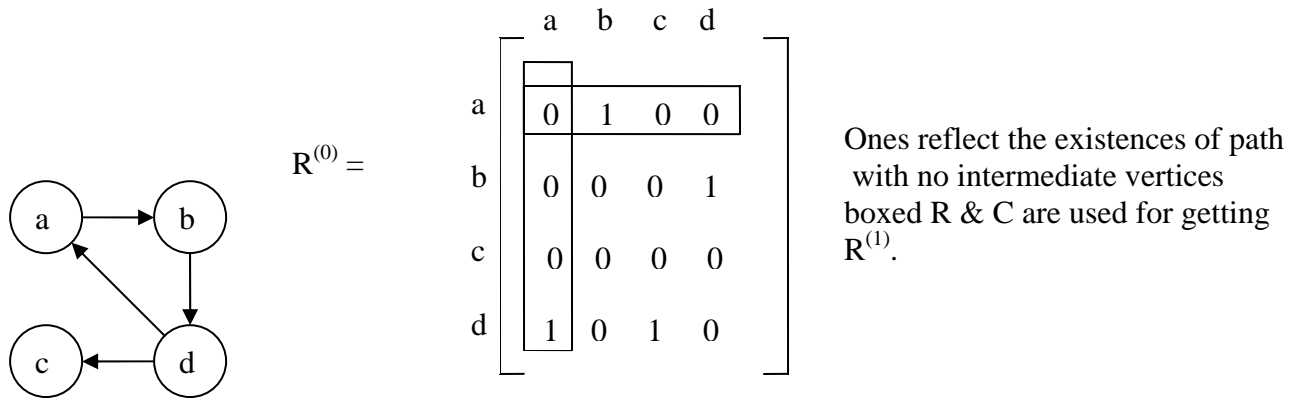
for $k \leftarrow 1$ to n do

 for $i \leftarrow 1$ to n do

 for $j \leftarrow 1$ to n do

$R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$

Return $R^{(n)}$.



$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

$R^{(2)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Intermediate is vertex a;

Intermediate vertex is a & b;

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)} =$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

Intermediate vertices are a, b & c

Intermediate vertices a, b, c & d

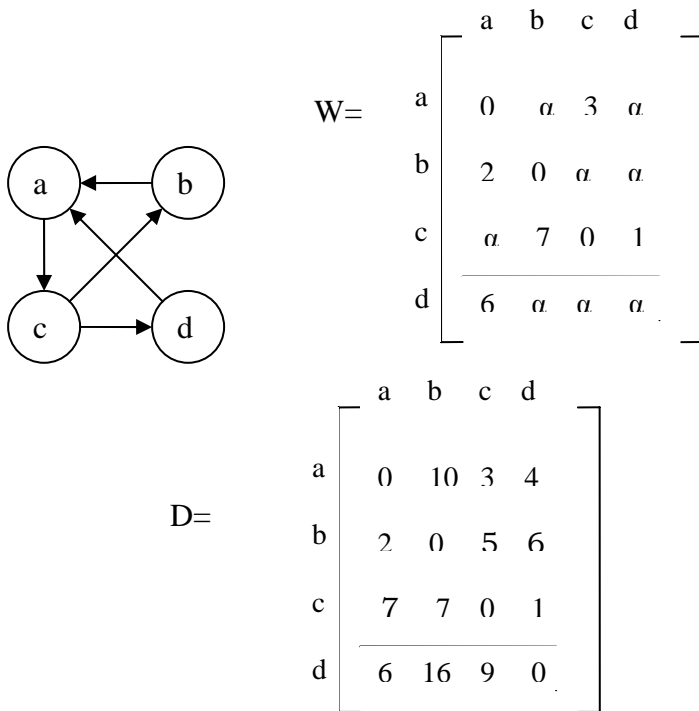
The time efficiency is in $\Theta(n^3)$. This can be speed up by restricting its innermost loop. Another way to make the algorithm run faster is to treat matrix rows as bit strings and apply bitwise OR operations.

As to the space is considered we used separate matrices for recording intermediate results, which is unnecessary.

Floyd's algorithm for the All-pairs shortest paths problem:

It is to find the distances (the length of the shortest paths) from each vertex to all other vertices. Its convenient to record the lengths of shortest paths in an $n \times n$ matrix D called distance matrix. The element d_{ij} in the i^{th} row and j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to j^{th} vertex ($1 \leq i, j \leq n$).

Floyd's algorithm is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of negative length.



It computes the distance matrix of a weighted graph with n vertices thru a series of $n \times n$ matrices.

$D^0, D^{(1)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}$

$D^{(0)}$ with no intermediate vertices and $D^{(n)}$ consists all the vertices as intermediate vertices. $d_{ij}^{(k)}$ is the shortest path from v_i to v_j with their intermediate vertices numbered not higher than k . $D^{(k)}$ is obtained from its immediate predecessor $D^{(k-1)}$.

i.e., v_i a list of int. vertices each numbered not higher than k , v_j .

We can partition the path into 2 disjoint subsets: those that do not use the k^{th} vertex as intermediate and those that do. Since the paths of the 1st subset have their intermediate vertices numbered not higher than $k-1$, the shortest of them is $d_{ij}^{(k-1)}$.

The length of the shortest path in the 2nd subset is, if the graph does not contain a cycle of negative length, the subset use vertex v_k as intermediate vertex exactly once.

v_i , vertices numbered $\leq k-1$, v_k , vertices numbered $\leq k-1$, v_j .

The path v_i to v_k is equal to $d_{ik}^{(k-1)}$ and v_k to v_j is $d_{kj}^{(k-1)}$, the length of the shortest path among the paths that use the k^{th} vertex is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \text{ for } k \geq 1,$$

$$d_{ij}^{(0)} = w_{ij}.$$

Algorithm Floyd($w[1..n, 1..n]$)

//Implement Floyd's algorithm for the all-pairs shortest-paths problem

// Input: The weight matrix W of a graph

// Output: The distance matrix of the shortest paths lengths

$D \leftarrow w$

For $k \leftarrow 1$ to n do

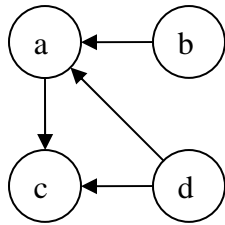
For $i \leftarrow 1$ to n do

For $j \leftarrow 1$ to n do

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

Return D .

The time efficiency is $\Theta(n^3)$. The principle of optimality holds for these problem where there is an optimization.

 $D^{(0)} =$

	a	b	c	d
a	0	α	3	α
b	2	0	α	α
c	α	7	0	1
d	6	α	α	0

Length with no intermediate vertices.

 $D^{(1)} =$

	a	b	c	d
a	0	α	3	α
b	2	0	5	α
c	α	7	0	1
d	6	α	9	0

Lengths of the path with intermediate vertex a;

 $D^{(2)} =$

	a	b	c	d
a	0	α	3	α
b	2	0	5	α
c	9	7	0	1
d	6	α	9	0

Lengths of the path with intermediate vertex a & b;

 $D^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	α
c	9	7	0	1
d	6	16	9	0

Lengths of the path with intermediate vertex a,b & c;

 $D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	α
c	7	7	0	1
d	6	16	9	0

Lengths of the path with intermediate vertex a,b,c & d;

8.3 The Knapsack problem and Memory functions:

The knapsack problem states that given n items of known weights w_1, w_2, \dots, w_n and values v_1, v_2, \dots, v_n and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack. In dynamic programming we need to obtain the solution by solving the smaller subinstances.

Let $v[i, j]$ be the value of an optimal soln to the instance, i.e., the value of the most valuable subset of the 1st i items that fit into the knapsack of capacity j . We can divide the subsets into 2: those that do not include the i^{th} item and those that do.

1. Among the subsets that do not include the i^{th} item, the value of an optimal subset is, by definition: $v[i-1, j]$
2. Among the subsets that do include the i^{th} item, an optimal subset is made up of this item and an optimal subset of the 1st $i-1$ items that fit into the knapsack of capacity $j-w_i$. The value of such an optimal subset is $v_i + v[i-1, j-w_i]$.

Thus the value of an optimal solution among all feasible subsets of the 1st i items is the maximum of these 2 values. Of course, if the i^{th} item does not fit into the knapsack, the value of an optimal subset selected from the 1st i items is the same as the value of an optimal subset selected from the first $i-1$ items.

$$\text{Therefore, } v[i, j] = \begin{cases} \max\{v[i-1, j], v_i + v[i-1, j-w_i]\}, & \text{if } j-w_i \geq 0 \\ v[i-1, j], & \text{if } j-w_i < 0. \end{cases}$$

The initial conditions are:

$$v[0, j] = 0 \text{ for } j \geq 0 \text{ and } v[i, 0] = 0 \text{ for } i \geq 0.$$

Our goal is to find $v[n, W]$, the maximal value of a subset of n items which fit into W .

		0	$j-w_i$	j	w
w_i, v_i	0	0	0	0	0
	$i-1$	0	$v[i-1, j-w_i]$	$v[i-1, j]$	
	i	0			
	n	0		$v[i, j]$	

Example:	items	weight	value	$W=5$
	1	2	\$12	
	2	1	\$10	
	3	3	\$20	
	4	2	\$15	

Apply the above formula to obtain the table:

		0	1	2	3	4	5
w_i, v_i	i	0	0	0	0	0	0
	1	0	0	12	12	12	12
	2	0	10	12	22	22	22
	3	0	10	12	22	30	32
	4	0	10	15	25	30	37

The maximal value is $v[4,5]=\$37$. Since $v[4,5] \neq v[3,5]$ item 4 was included in an optimal solution along with an optimal subset for filling $5-2 = 3$ remaining units of the knapsack capacity. Next $v[3,3] = v[2,3]$, item 3 is not a part of an optimal subset. Since $v[2,3] \neq v[1,3]$ item 2 is a part of an optimal selection, which leaves $elt.v[1,3-1]$ to specify the remaining composition. $v[1,2] \neq v[0,2]$ item 1 is part of the solution. Therefore $\{item1,item2,item4\}$ is the subset with value 37.

Memory Functions:

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient. The other approach is bottom-up, it fills a table with solutions to all smaller subproblems but each of them is solved only once. The drawback of bottom-up is to solve all smaller subproblems even if its not necessary for getting the solution we use to combine the top-down and bottom-up, where the goal is to get a method that solves only subproblems that are necessary and does it only once. Such a method is by using memory functions.

This method solves by top-down; but, creates a table to be filled in by bottom-up. Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated called virtual initialization. Therefore, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not null its retrieved from the table; otherwise, its computed by the recursive call whose result is then recorded in the table.

Example:

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0	0	12	-	12	12
$w_2=1, v_2=10$	2	0	-	12	22	-	22
$w_3=3, v_3=20$	3	0	-	-	22	-	32
$w_4=2, v_4=15$	4	0	-	-	-	-	37

Algorithm MF Knapsack(i,j)

// Uses a global variables input arrays $w[1..n], v[1..n]$ and

// Table $v[0..n, 0..w]$ whose entries are initialized

// with -1's except for row 0 and column 0 initialized with 0's.

if $v[i,j] < 0$

 if $j < \text{weights}[i]$

 value \leftarrow MF knapsack(i-1, j)

 else

 value \leftarrow max (MF knapsack(i-1,j), values[i]+MF knapsack(i-1,j-weights[i]))

$v[i,j] \leftarrow$ value

 return $v[i,j]$.

Chapter 9. Greedy Technique

Introduction:

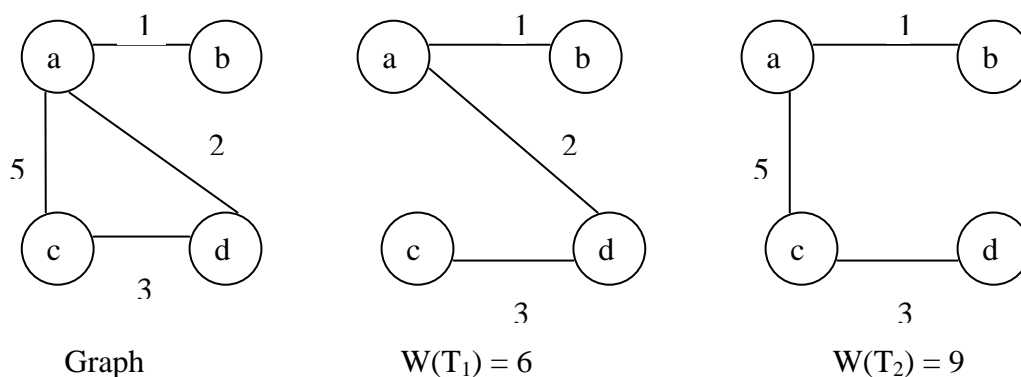
The change making problem is a good example of greedy concept. That is to give change for a specific amount 'n' with least number of coins of the denominations $d_1 > d_2 > \dots > d_m$. For example: $d_1=25$, $d_2=10$, $d_3=5$, $d_4=1$. To get a change for 48 cents. First step is to give 1 d_1 , 2 d_2 and 3 d_4 's which gives a optimal solution with the lesser number of coins. The greedy technique is applicable to optimization problems. It suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be -

- Feasible - i.e., it has to satisfy the problem constraints.
- Locally optimal - i.e., it has to be the best local choice among all feasible choices available on that step.
- Irrevocable - i.e., once made, it cannot be changed on subsequent steps of the algorithm.

9.1 Prim's algorithm:

Definition: - A spanning tree of connected graph is its connected acyclic sub graph that contains all the vertices of the graph. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of the tree is defined as the sum of the weight on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Two serious obstacles are: first, the number of spanning tree grows exponentially with the graph size. Second, generating all spanning trees for the given graph is not easy; in fact, it's more difficult than finding a minimum spanning for a weighted graph by using one of several efficient algorithms available for this problem.



Graph and its spanning tree ; T_1 is the min spanning tree .

Prim's algorithm constructs a minimum spanning tree thru a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after being constructed. The total number of iterations will be $n-1$, if there are 'n' number of edges.

Algorithm Prim(G)

// Input: A weighted connected graph $G = \{V, E\}$

// Output: E_T , the set of edges composing a minimum spanning tree of G .

$V_T \leftarrow \{V_0\}$

$E_T \leftarrow \emptyset$

For $i \leftarrow 1$ to $|V|-1$ do

Find a minimum weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
such that v is in V_T & u is in $V - V_T$.

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

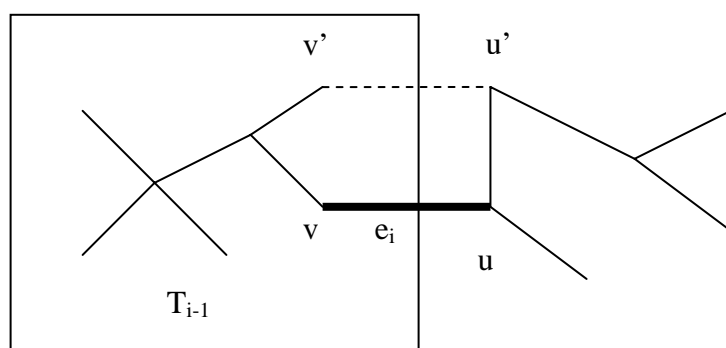
Return E_T .

The algorithm makes to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. Vertices that are not adjacent is labeled " ∞ " (infinity). The Vertices not in the tree are divided into 2 sets : the fringe and the unseen. The fringe contains only the vertices that are not in the tree but are adjacent to atleast one tree vertex. From these next vertices is selected. The unseen vertices are all the other vertices of the graph, (they are yet to be seen). This can be broken arbitrarily.

After we identify a vertex u^* to be added to the tree, we need to perform 2 operations:

- Move u^* from the set $V - V_T$ to the set of tree vertices V_T .
- For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u 's current distance label, update its label by u^* and the weight of the edge between u^* & u , respectively.

Prim's algorithm yields always a Minimum Spanning Tree. The proof is by induction. Since T_0 consists of a single vertex and hence must be a part of any Minimum Spanning Tree. Assume that T_{i-1} is part of some Minimum Spanning Tree. We need to prove that T_i generated from T_{i-1} is also a part of Minimum Spanning Tree, by contradiction. Let us assume that no Minimum Spanning Tree of the graph can contain T_i . let $e_i = (v, u)$ be the minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i , e_i cannot belong to any Minimum Spanning Tree including T . Therefore if we add e_i to T , a cycle must be formed.



In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex $v' \in T_{i-1}$ to a vertex u' which is not in T_{i-1} . If we now delete the edge (v', u') from this cycle we obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T . Since the weight of e_i is less than or equal to the weight of (v', u') . Hence this is a Minimum Spanning Tree and contradicts that no Minimum Spanning Tree contains T_i .

The efficiency of this algorithm depends on the data structure chosen for the graph itself and for the priority queue of the set $V - V_T$ whose vertex priorities are the distance to the nearest tree vertices. For eg, if the graph is represented by weight

matrix and the priority queue is implemented as an unordered array the algorithm's running time will be in $\Theta(|V|^2)$.

Priority queue can be implemented by a min_heap, which is a complete binary tree in which every element is less than or equal to its children. Deletion of smallest element from and insertion of a new element into a min_heap of size n is $O(\log n)$ operations.

If a graph is represented by its adjacency linked lists and the priority queue is implemented as a min_heap, the running time of the algorithm's is in $O(|E|\log|V|)$. This is because the algorithm performs $|V|-1$ deletions of the smallest element and makes $|E|$ verifications, and changes of an element's priority in a min_heap of size not greater than $|V|$. Each of these operations is a $O(\log|V|)$ operations. Hence, the running time is in:

$$(|V|-1+|E|) O(\log|V|) = O(|E|\log|V|)$$

Because, in a connected graph, $|V|-1 \leq |E|$.

9.2 Kruskal's algorithm:

It looks at Minimum Spanning Tree for a weighted connected graph $G=\langle V,E \rangle$ as an acyclic sub graph with $|V|-1$ edges for which the sum of the edges weights is the smallest. The algorithm constructs a Minimum Spanning Tree as an expanding sequence of sub graphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graphs edges in increasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skips the edge otherwise.

Algorithm Kruskal(G)

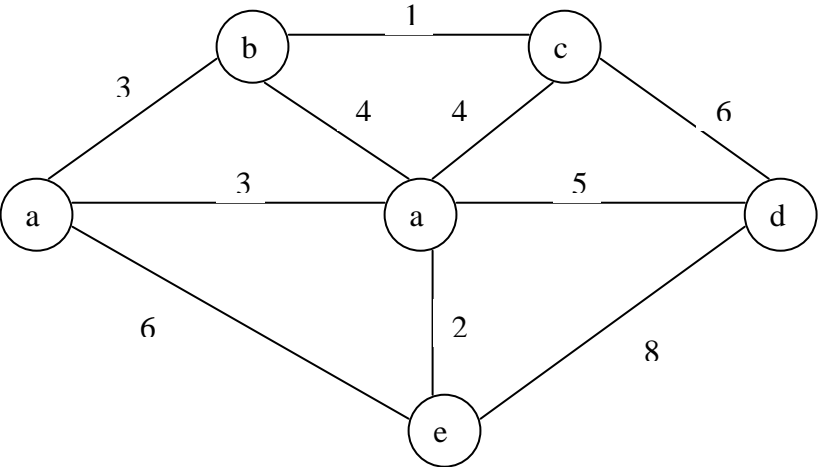
// Input: A weighted graph $G=\langle V,E \rangle$

// Output: E_T , the set of edges composing a Minimum Spanning Tree of G

Sort E in increasing order of edge weights

$ET \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size.

```
K <- 0 //initialize the no of processed edges
While ecounter < |V|-1
    K <- K+1
    if ET U{eik} is acyclic
        ET <- ET U {eik};
        ecounter<-ecounter+1
Return ET.
```



Tree edge	Sorted list of edges
	bc ef ab bf cf af df ac 1 2 3 4 4 5 5 6
bc 1	bc ef ab 1 2 3
ef 2	bc ef ab 1 2 3
ab 3	bc ef ab bf..... 1 2 3 4
bf 4	bc ef ab bf df..... 1 2 3 4 5 (cf & af are not chosen, because it forms cycle)

Kruskal's algorithm is not simpler than prim's. Because , on each iteration it has to check whether the edge added forms a cycle. Each connected component of a sub graph generated is a tree because it has no cycles.

There are efficient algorithms for doing these observations, called union_find algorithms. With this the time needed for sorting the edge weights of a given graph and it will be $O(|E|\log|E|)$.

Disjoint sub sets and union_find algorithm.

Kruskal's algorithm requires a dynamic partition of some n-element set S into a collection of disjoint subsets $S_1, S_2, S_3, \dots, S_k$. After initializing each consist of different elements of S , which is the sequence of intermixed union and find algorithms or operations . Here we deal with an abstract data type of collection of disjoint subsets of a finite set with the following operations:

- **Makeset(x):** Creates a 1-elt set $\{x\}$. Its assumed that this operations can be applied to each of the element of set S only once.
- **Find(x):** Returns a subset containing x .
- **Union(x,y):** Constructs the union of the disjoint subsets S_x & S_y containing x & y respectively and adds it to the collection to replace S_x & S_y , which are deleted from it.

For example , Let $S=\{1,2,3,4,5,6\}$. Then make(i) Creates the set $\{i\}$ and apply this operation to create single sets:

$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$.

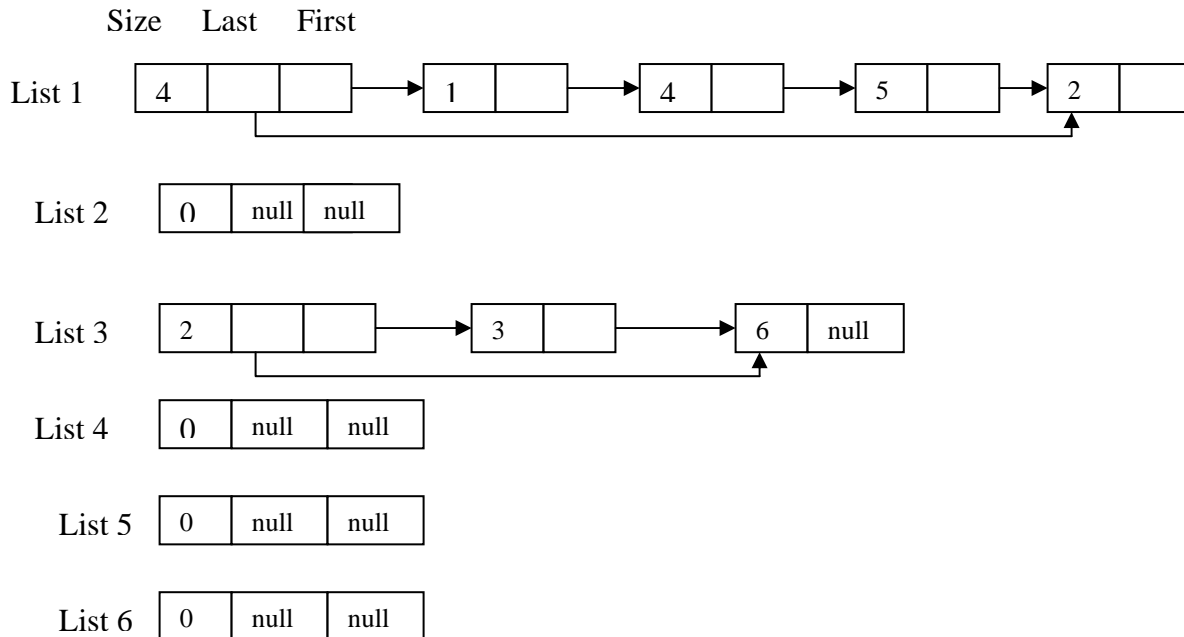
Performing union (1,4) & union(5,2) yields

$\{1,4\}, \{5,2\}, \{3\}, \{6\}$

Then union(4,5) & union(3,6) gives

$\{1,4,5,2\}, \{3,6\}$

It uses one element from each of the disjoint sub sets in a collection as that subset's representative. There are two alternatives for implementing this data structure, called the quick find, optimizes the time efficiency of the find operation, the second one, called the quick union, optimizes the union operation.



Subset representatives	
Element Index	Representation
1	1
2	1
3	3
4	1
5	1
6	3

For the makeset(x) time is in $\Theta(1)$ and for 'n' elements it will be in $\Theta(n)$. The efficiency of find(x) is also in $\Theta(1)$: union(x,y) takes longer of all the operations like update & delete.

Union (2,1), union(3,2),..... , union(i+1,i),.....union(n,n-1)

Which runs in $\Theta(n^2)$ times , which is slow compared to other ways: there is an operation called union-by-size which unites based on the length of the list. i.e., shorter of the two list will be attached to longer one, which takes $\Theta(n)$ time. Therefore, the total time needed will be $\Theta(n \log n)$.

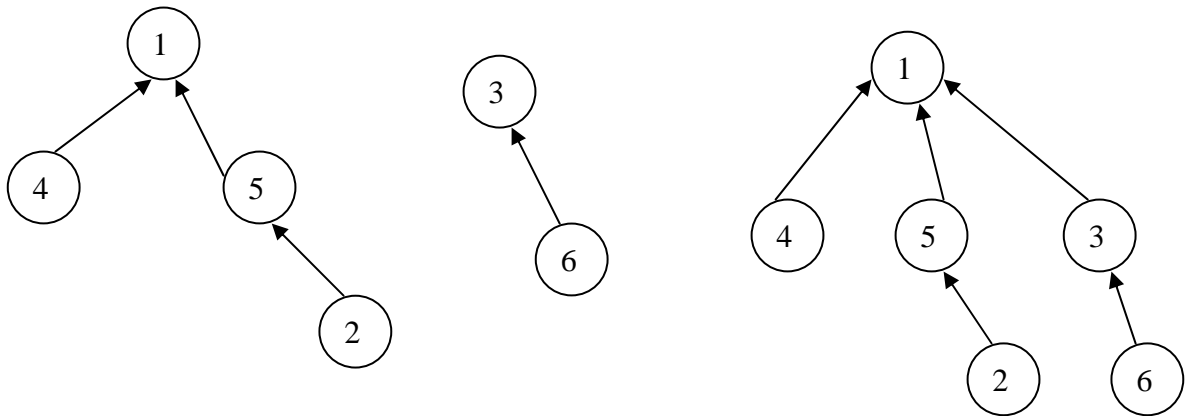
Each a_i is updated A_i times, the resulting set will have 2^{A_i} elements. Since the entire set has n elements, $2^{A_i} \leq n$ and hence $A_i \leq \log_2 n$. Therefore, the total number of possible updates of the representatives for all n elements is S will not exceed $n \log_2 n$.

Thus for union by size, the time efficiency of a sequence of at most $n-1$ unions and m finds is in $O(n \log n + m)$.

The quick union-represents by a rooted tree. The nodes of the tree contain the subset elements, with the roots element considered the subsets representatives, the tree's edges are directed from children to their parents.

Makeset(x) requires $\Theta(1)$ time and for ' n ' elements it is $\Theta(n)$, a union(x, y) is $\Theta(1)$ and find(x) is in $\Theta(n)$ because a tree representing a subset can degenerate into linked list with n nodes.

The union operation is to attach a smaller tree to the root of a larger one. The tree size can be measured either by the number of nodes or by its height(union-by-rank). To execute each find it takes $O(\log n)$ time. Thus, for quick union, the time efficiency of at most $n-1$ unions and m finds is in $O(n + m \log n)$.



Forest representation of subsets $\{1, 4, 5, 2\}$ & $\{3, 6\}$.
Resultant of union($5, 6$).

An even better efficiency can be obtained by combining either variety of quick union with path compression. This modification makes every node encountered during the execution of a find operation point to the tree's root.

9.3 Dijkstra's Algm:

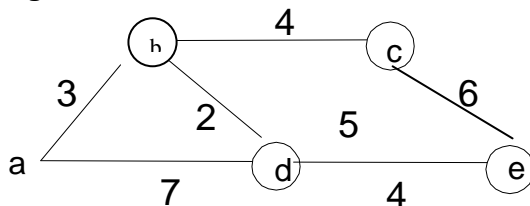
Here, we consider the single-source shortest paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices. It is a set of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

This algorithm is applicable to graphs with nonnegative weights only. It finds shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest and so on. In general, before its i^{th} iteration commences, the algorithm has already identified its shortest path to $i-1$ other vertices nearest to the source. This forms a sub tree T_i and the next vertex chosen to be should be vertices adjacent to the vertices of T_i , fringe vertices. To identify, the i^{th} nearest vertex, the algorithm computes for every fringe vertex u , the sum of the distance to the nearest tree vertex v and then select the smallest such sum. Finding the next nearest vertex u^* becomes the simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

- Move u^* from the fringe to the set of the tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ s.t. $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$ respectively.

Eg.



Tree vertices	Remaining vertices	Illustration
a(-,0)	b(a,3), c(-,∞), d(a,7), e(-,∞)	
b(a,3)	c(b,3+4), d(b,3+2), e(-,∞)	
d(b,5)	c(b,7), e(d,5+4)	
c(b,7)	e(d,9)	
e(d,9)		

The shortest paths and their lengths are:

From a to b: a-b of length 3

From a to d: a-b-d of length 5

From a to c: a-b-c of length 7

From a to e: a-b-d-e of length 9

The next vertex chosen is from the priority queue of the remaining vertices. The algorithm compares path lengths and therefore must add edge weights, while prim's algorithm compares the edge weights as given.

Algorithm Dijkstra(a)

// Input: A weighted connected graph $G=\langle V,E \rangle$ and its vertex s

// Output: The length d_v of a shortest path from s to v and its penultimate vertex p_v //

// for every vertex v in V .

```

Initialize(Q) //initialize vertex priority queue to empty for every vertex v in V do
for every vertex v in V do
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
    Insert(Q,v,  $d_v$ ) //initialize vertex priority in the priority queue
 $d_s \leftarrow 0$ ; decrease(Q,s,  $d_s$ ) //update priority of s with  $d_s$ 
 $V_T \leftarrow \emptyset$ 
for i=0 to  $|V|-1$  do
     $u^* \leftarrow \text{deleteMin}(Q)$  //delete the minimum priority element
     $V_T \leftarrow V_T \cup \{u^*\}$ 
    for every vertex u in  $V - V_T$  that is adjacent to  $u^*$  do
        if  $d_{u^*} + w(u^*,u) < d_u$ 
             $d_u \leftarrow d_{u^*} + w(u^*,u)$ ;  $p_u \leftarrow u^*$ 
            Decrease(Q,u,  $d_u$ )

```

The time efficiency depends on the data structure used for implementing the priority queue and for representing the input graph. It is in $\Theta(|V|)^2$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency linked lists and the priority queue implemented as a min_heap it is in $O(|E|\log|V|)$.

9.4 Huffman Trees

Suppose we have to encode a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bits called the code word. Two types are there :fixed length encoding that assigns to each character a bit string of the same length m variable-length encoding, which assigns codewords of different lengths to different characters , introduces a problem that fixed length encoding does not have.

To avoid complication, we called prefix-free code. In a prefix code, no codeword is a prefix of a code word of another character. Hence, with such an encoding we can simplify scan a bit string until we get the first group of bits that is a code word for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

Huffman's algorithm

Step 1: Initialize n one-node trees and label them with the characters of the alphabet.
Record the frequency of each character in its tree's root to indicate the tree's

weight.

Step 2: Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right sub tree of a new tree and record the sum of their weights in the root of the new tree as its weight. A tree constructed by the above algm is called a Huffman tree. It defines - a Huffman code.

Example : consider the 5 char alphabet {A,B,C,D,-} with the following occurrence probabilities:

Char	A	B	C	D	—
Probability	0.35	0.1	0.2	0.2	0.15

The resulting code words are as follows:

Char	A	B	C	D	—
Probability	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101

Hence DAD is encoded as 011101, and 10011011011101 is decoded as BAD-AD.

With the occurrence probabilities given and the codeword lengths obtained, the expected number of bits per char in this code is:

$$2*0.35 + 3*0.1 + 2*0.2 + 2*0.2 + 3*0.15 = 2.25$$

The compression ratio, is a measure of the compression algorithms effectiveness of $(3-2.25)/3*100\%=25\%$. The coding will use 25% less memory than its fixed length encoding.

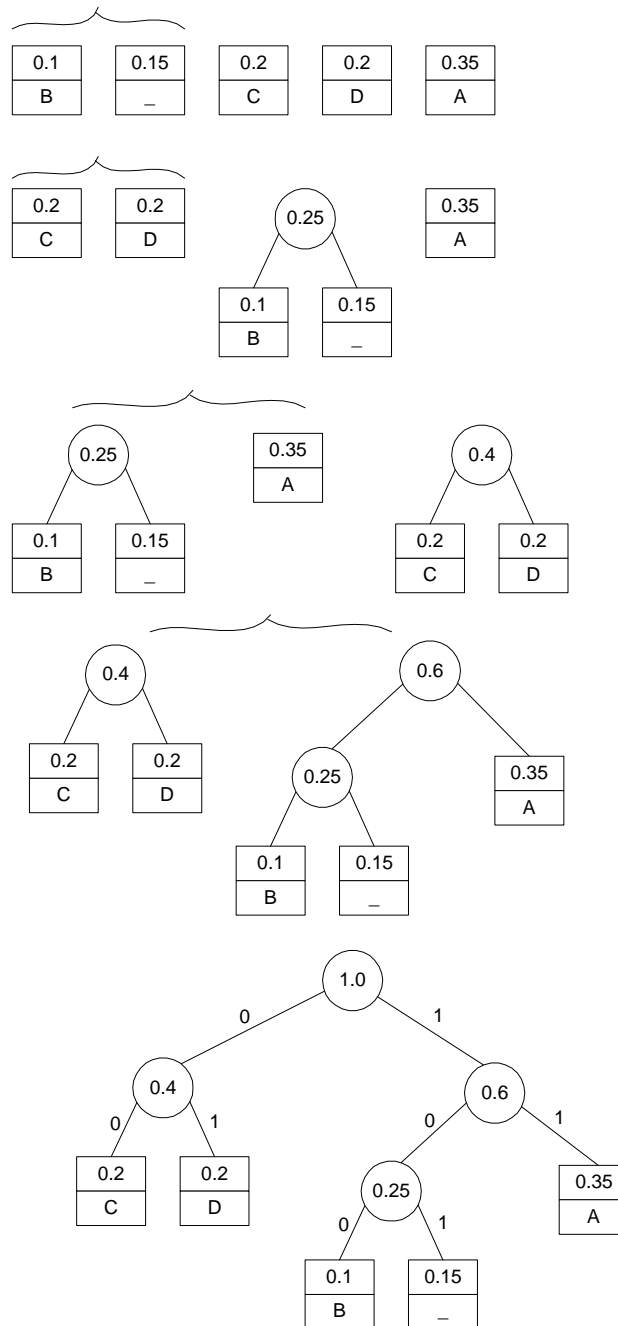
Huffman's encoding is one of the most important file compression methods. It is simple and yields an optimal.

The draw back can be overcome by the so called dynamic Huffman encoding, in which the coding tree is updated each time a new char is read from source text.

n

Huffman's code is not only limited to data compression. The sum $\sum_{i=1} l_i w_i$ where

l_i is the length of the simple path from the root to i^{th} leaf, it is weighted path length. From this decision trees, can be obtained which is used for game applications.



Chapter 10. Limitations of Algorithm power

10.1 Lower-Bound Arguments:

Two ways to determine the efficiency of an algorithm is to know the asymptotic efficiency class and the class of hierarchy. The alternative is to find how efficient a particular algorithm is with respect to other algorithms for the same problem. When we want to ascertain the efficiency of an algorithm with respect to other algorithms for the same problem, it is desired to know the best possible efficiency of any algorithm solving the problem may have. Knowing such a lower bound can give us the improvement to achieve an efficient algorithm. If such a bound is tight, i.e., if we already know an algorithm in the same efficiency class as the lower bound, we can hope for a constant-factor improvement at best.

Trivial Lower Bounds is used to yield the bound best option is to count the number of items in the problem's input that must be processed and the number of output items that need to be produced.

Information theoretic Arguments seeks to establish a lower bound based on the amount of information it has to produce. It is called so because of its connection to information theory. The decision trees are the mechanism used to solve the problem.

Adversary arguments establish the lower bounds. Problem reduction is used based on the other problem where an efficient algorithm is solved.

10.2 Decision Trees:

The number of leaves must be at least as large as the number of possible outcomes. The algorithm's work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm. On such a run is equal to the number of edges in this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree. It is used for comparison-based sorting algorithm. Decision trees are also used for searching a sorted array, which can be either a 2-node or 3-node trees.

10.3 P, NP, and NP-complete problems:

Tractable problems are that can be solved in polynomial time and nontractable are those which cannot be solved in polynomial time. Decision problems are also a p problems where a decision can be taken.

A decision problem that can be solved in polynomial time is called polynomial. e.g., m-coloring problem. Undesired problems cannot be solved by any algorithm. Halting problem will halt on that on that input or continue working indefinitely on it. Nondeterministic polynomial is the class of decision problems that can be solved by nondeterministic algorithms.

$$P \subseteq NP$$

Non-deterministic consists of 2 stages. First is the guessing stage, an arbitrary string S is generated that can be thought of as a candidate solution to the given instance I . Second stage is verification stage, a deterministic algorithm takes both I & S as its input and output yes if S represents a solution to instance I . NP-complete problem is a problem in NP that is as difficult as any other problem in this class because, by definition, any other problem in NP can be reduced to it in polynomial time.

A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that

- 1) t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2 ;
- 2) t is computed by a polynomial time algorithm.

A decision problem is said to be NP-complete if

- 1) it belongs to class NP;
- 2) every problem in n NP is polynomially reducible to D.

CNF-satisfiability is NP-complete. It deals with Boolean expressions.

eg: $(x_1 \vee x_2 \vee x_3) \& (x_1 \vee x_2)$

Chapter 11. Coping with the Limitations of Algorithm Power

Introduction:

Backtracking & Branch-and-Bound are the two algorithm design techniques for solving problems in which the number of choices grows at least exponentially with their instance size. Both techniques construct a solution one component at a time trying to terminate the process as soon as one can ascertain that no solution can be obtained as a result of the choices already made. This approach makes it possible to solve many large instances of NP-hard problems in an acceptable amount of time.

Both Backtracking and branch-and-bound uses a state-space tree-a rooted tree whose nodes represent partially constructed solutions to the problem. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants.

11.1 Backtracking:

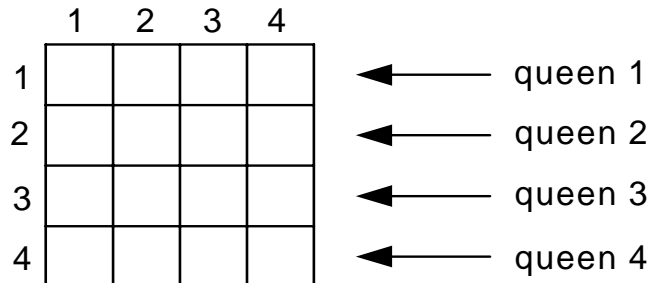
Backtracking constructs its state-space tree in the depth-first search fashion in the majority of its applications. If the sequence of choices represented by a current node of the state-space tree can be developed further without violating the problem's constraints, it is done by considering the first remaining legitimate option for the next component. Otherwise, the method backtracks by undoing the last component of the partially built solution and replaces it by the next alternative.

A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise it is called non promising. Leaves represent either nonpromising dead ends or complete solutions found by the algorithms.

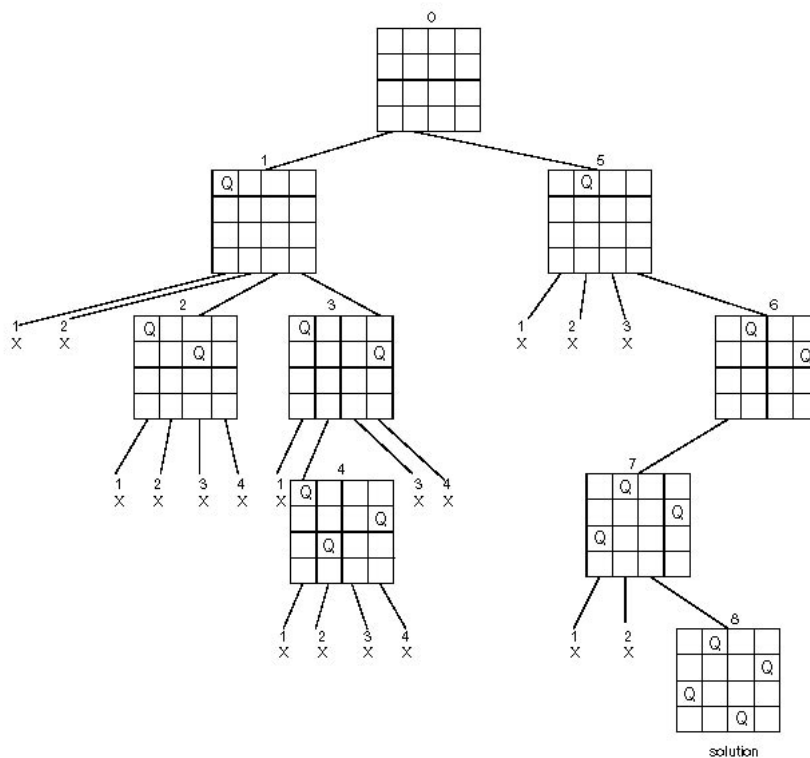
n-Queens problem:

The problem is to place n Queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

Place n queens on an n -by- n chess board so that no two of them are in the same row, column, or diagonal

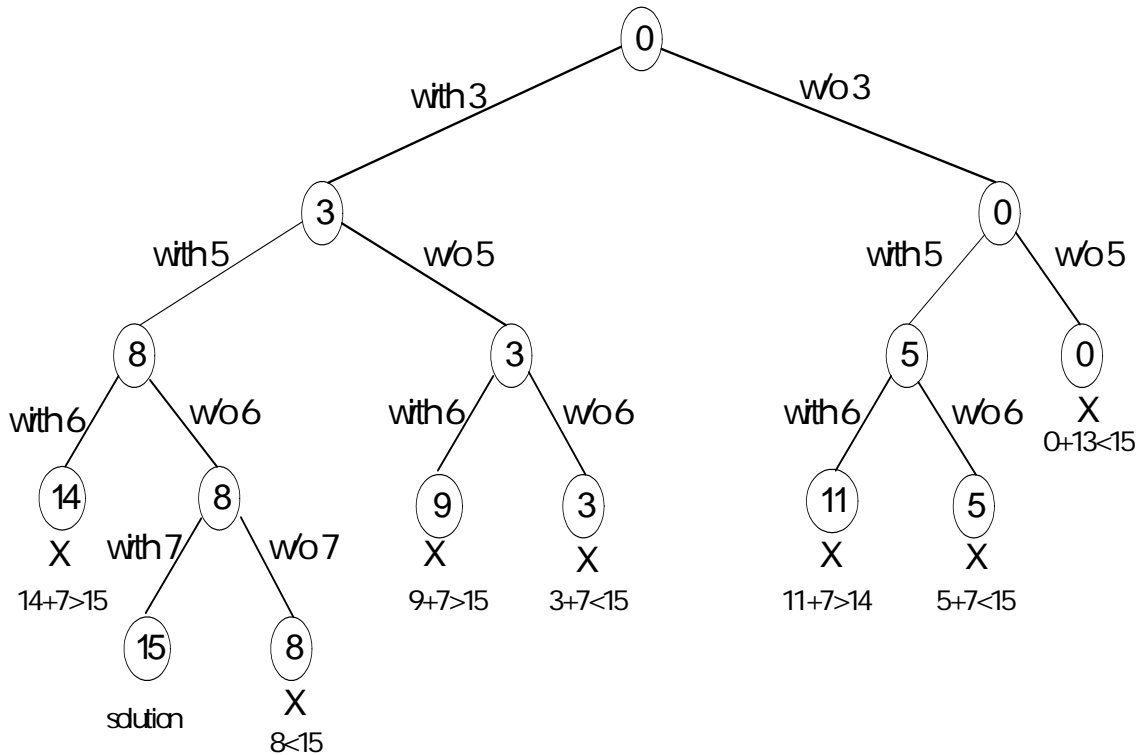
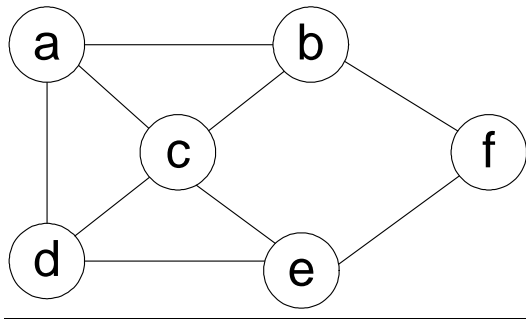


State-Space Tree of the 4-Queens Problem



Hamiltonian Circuit problem:

Assume that if a Hamiltonian circuit exists, it starts at vertex a . Then vertex a will be the root of the state-space tree, and, then from vertex a traverse thru all vertices without forming a cycle.

Subset-Sum problem:

Find a subset of a given set $s=\{s_1, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d . For ex, $s=\{1,2,5,6,8\}$ and $d=9$, there are 2 solutions, $\{1,2,6\}$ & $\{1,8\}$. It is convenient to sort the elements in increasing order:

$$s_1 \leq s_2 \leq s_3 \dots \dots \dots s_n$$

We record the value of s' , the sum of these numbers in the node. If s' is equal to d , we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by

backtracking to the node's parent. If s' is not equal to d , we can terminate the node as nonpromising if either of the two inequalities holds:

$$s' + s_{i+1} > d \text{ (the sum } s' \text{ is too large)}$$

$$s' + \sum_{j=i+1}^n s_j < d \text{ (the sum } s' \text{ is too small)}$$

It is generally used as 'n' tuples to generate the tree as $x_1, x_2, x_3, \dots, x_n$

Algorithm Backtrack ($x[1..i]$)

//Output: All the tuples representing the problem's solutions

if $x[1..i]$ is a solution write $x[1..i]$

else

for each element $x \in s_{i+1}$ consistent with $x[1..i]$ and the constraints do

$x[i+1] \leftarrow x$

Backtrack ($x[1..i+1]$)

In conclusion, the backtracking first is typically applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist. Second, unlike the exhaustive search approach, which is extremely slow, backtracking atleast for some problem it gives in a acceptable amount of time, which is usually in the cases of optimization problems. Third, in backtracking doesn't eliminate any elements of a problem's state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of value in its own right.

11.2 Branch-and-Bound:

It is an algorithm design technique that enhances the idea of generating a state-space tree with the idea of estimating the best value obtainable from a current node of the decision tree: if such an estimate is not superior to the best solution seen up to that point in the processing, the node is eliminated from further consideration.

A feasible solution, is a point in the problem's search space that satisfies all the problem's constraints, while an optimal solution is a feasible

solution with the best value of the objective function compared to backtracking branch-and-bound requires 2 additional items:

- 1) A way to provide, for every node of a state-space tree a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partial solution represented by the node.
- 2) The best value of the best solution seen so far.

If this information is available, we can compare a node's bound with the value of the best solution seen so far: if the bound value is not better than the best solution seen so far- i.e., not smaller for a minimization problem and not larger for a maximization problem- the node is nonpromising and can be terminated because no solution obtained from it can yield a better solution than the one already available.

In general, we terminate a search path at the current node in a state-space tree of a branch & bound algorithm for any one of the following three reasons:

- 1) The value of the node's bound is not better than the value of the best solution seen so far.
- 2) The node represents no feasible solutions because the constraints of the problem are already violated.
- 3) The subset of feasible solutions represented by the node consists of a simple point. Compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Assignment problem:

It is a problem where each job will be assigned to each person. And no 2 jobs can be assigned to same person and no 2 person should be assigned with the same job.

Select one element in each row of the cost matrix C so that:

- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person <i>a</i>	9	2	7	8
Person <i>b</i>	6	4	3	7
Person <i>c</i>	5	8	1	8
Person <i>d</i>	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: $2 + 3 + 1 + 4$ (or $5 + 2 + 1 + 4$)

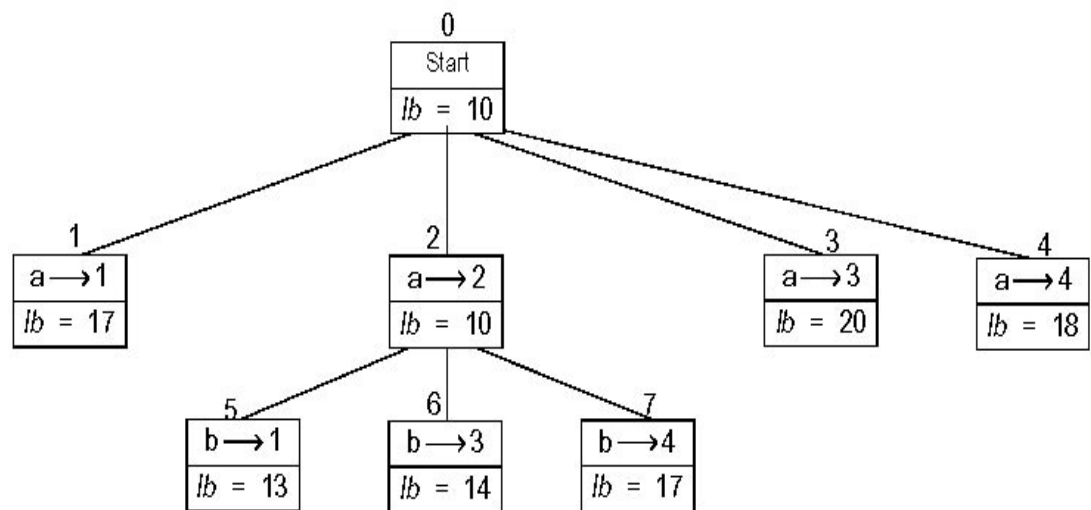


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

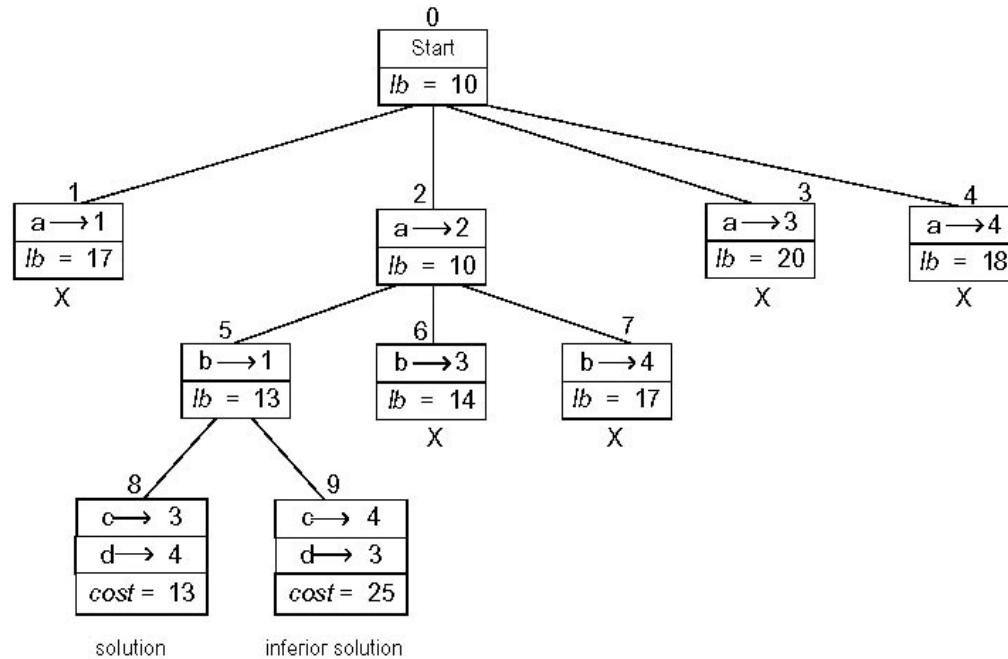


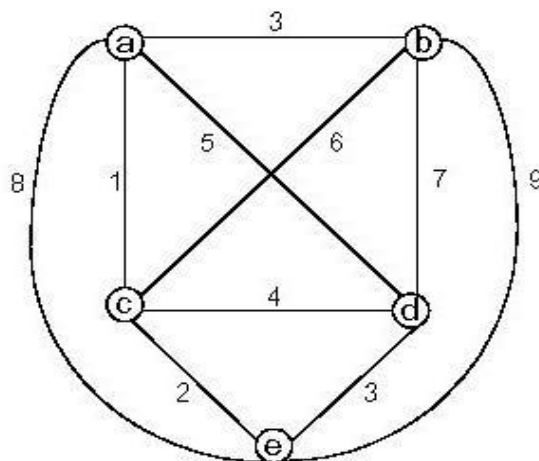
Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Knapsack problem:

Given n items of known weights W_i and values V_i , $i=1,2,\dots,n$ and a knapsack of capacity W , find the most valuable subset of the items that fit in the knapsack.

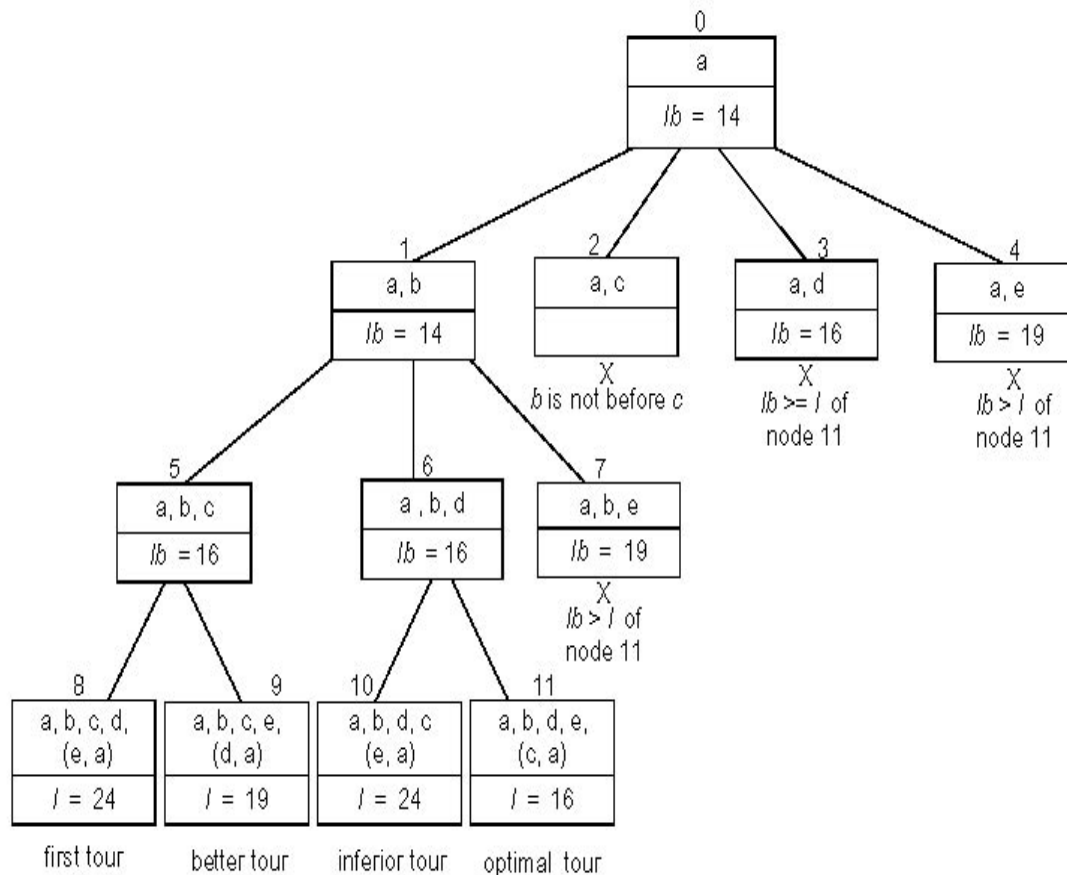
Traveling salespersons problem:

Visit all the vertices with a low cost yields the optimal solution.



Weighted graph

State-Space tree with the list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node



The lower bound is obtained as $lb = s/2$; where s is the sum of the distance of the n cities.

i.e., $[[(1+5) + (3+6) + (1+2) + (3+5) + (2+3)] / 2] = 16$, which is the optimal tour.

11.3 Approximation Algorithms for NP-hard problems:

It is the combinatorial problems which fall under NP-Hard problems. Accuracy ratio and performance ratio has to be calculated. Nearest-neighbour algorithm and Twice-around-the-tree algorithm. Greedy algorithm is used for the continuous knapsack problem for the fractional version.

Approximation algorithms are often used to find approximation solutions to difficult problems of combinatorial optimization. The

performance ratio is the principal metric for measuring the accuracy of such approximation algorithms.

Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it

Accuracy measures:

accuracy ratio of an approximate solution sa

$r(sa) = f(sa) / f(s^*)$ for minimization problems

$r(sa) = f(s^*) / f(sa)$ for maximization problems

where $f(sa)$ and $f(s^*)$ are values of the objective function f for the approximate solution sa and actual optimal solution s^* , performance ratio of the algorithm A the lowest upper bound of $r(sa)$ on all instances.

The nearest-neighbor is a greedy method for approximating a solution to the traveling salesman problem. The performance ratio is unbounded above, even for the important subset of Euclidean graphs.

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one

Note: Nearest-neighbor tour may depend on the starting city

Accuracy: $RA = \infty$ (unbounded above) - make the length of AD arbitrarily large in the above example

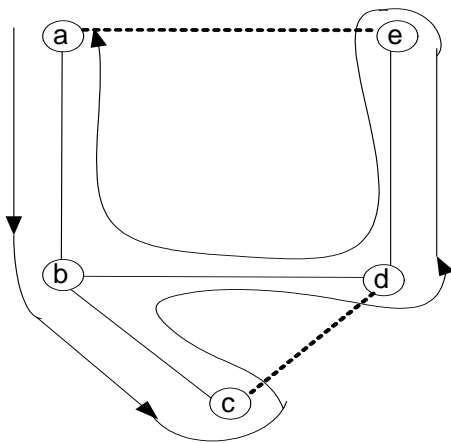
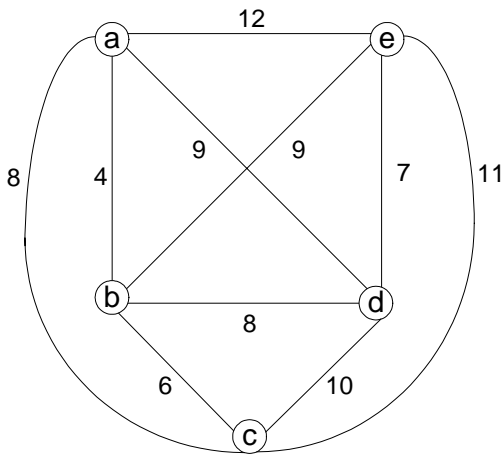
Twice-around-the-tree is an approximation algorithm for TSP with the performance ratio of 2 for Euclidean graph. The algorithm is based on modifying a walk around a MST by shortcuts.

Stage 1: Construct a minimum spanning tree of the graph (e.g., by Prim's or Kruskal's algorithm)

Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex

Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: $RA = \infty$ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm



Walk: a - b - c - b - d - e - d - b - a

Tour: a - b - c - d - e - a

A sensible greedy algorithm for the knapsack problem is based on processing an input's items in descending order of their value-to-weight ratios. For continuous version, the algorithm always yields an exact optimal solution.

Greedy Algorithm for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:

$$v_1/w_1 \geq \dots \geq v_n/w_n$$

Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	\$40	20
2	5	\$30	6
3	10	\$50	5
4	5	\$10	2

Accuracy

- ∅ RA is unbounded (e.g., $n = 2$, $C = m$, $w_1=1$, $v_1=2$, $w_2=m$, $v_2=m$)
- ∅ yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:

$$v_1/w_1 \geq \dots \geq v_n/w_n$$

Step 2: For a given integer parameter k , $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios

Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

- Accuracy: $f(s^*) / f(sa) \leq 1 + 1/k$ for any instance of size n
- Time efficiency: $O(kn^{k+1})$
- There are *fully polynomial schemes*: algorithms with polynomial running time as functions of both n and k

Polynomial approximation schemes for the knapsack problem are polynomial time parametric algorithms that approximation solutions with any predefined accuracy level.
