M.I.E.T. ENGINEERING COLLEGE

(Approved by AICTE and Affiliated to Anna University Chennai)

TRICHY – PUDUKKOTTAI ROAD, TIRUCHIRAPPALLI – 620 007

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIAL

CS6402 DESIGN AND ANALYSIS OF ALGORITHMS

II YEAR - IV SEMESTER

M.I.E.T. ENGINEERING COLLEGE

(Approved by AICTE and Affiliated to Anna University Chennai)

TRICHY – PUDUKKOTTAI ROAD, TIRUCHIRAPPALLI – 620 007

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SYLLABUS (THEORY)

Sub. Code : CS6402 **Branch / Year / Sem** : CSE/II/IV
Sub.Name : DESIGN AND ANALYSIS OF ALGORITHMS **Staff Name** : P.CHRISTOPHER

CS6402	DESIGN AND ANALYSIS OF ALGORITHMS	L T P C
		3 0 0 3

UNIT I INTRODUCTION 9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types –Fundamentals of the Analysis of Algorithm Efficiency – Analysis Framework – Asymptotic Notations and its properties – Mathematical analysis for Recursive and Non-recursive algorithms.

UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER 9

Brute Force - Closest-Pair and Convex-Hull Problems-Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and conquer methodology – Merge sort – Quick sort – Binary search – Multiplication of Large Integers – Strassen's Matrix Multiplication-Closest-Pair and Convex-Hull Problems.

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE 9

Computing a Binomial Coefficient – Warshall's and Floyd' algorithm – Optimal Binary Search Trees –Knapsack Problem and Memory functions. Greedy Technique– Prim's algorithm-Kruskal's Algorithm- Dijkstra's Algorithm-Huffman Trees.

UNIT IV ITERATIVE IMPROVEMENT 9

The Simplex Method-The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs-The Stable marriage Problem.

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER 9

Limitations of Algorithm Power-Lower-Bound Arguments-Decision Trees-P, NP and NP-Complete Problems--Coping with the Limitations - Backtracking – n-Queens problem – Hamiltonian Circuit Problem – Subset Sum Problem-Branch and Bound – Assignment problem – Knapsack Problem – Travelling Salesman Problem- Approximation Algorithms for NP – Hard Problems – Travelling Salesman problem – Knapsack problem.

TOTAL: 45 PERIODS

TEXT BOOK:

1. Anany Levitin, "Introduction to the Design and Analysis of Algorithms", Third Edition, Pearson Education, 2012.

REFERENCES:

1. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Third Edition, PHI Learning Private Limited, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "Data Structures and Algorithms", Pearson Education, Reprint 2006.
3. Donald E. Knuth, "The Art of Computer Programming", Volumes 1& 3 Pearson Education, 2009. Steven S. Skiena, "The Algorithm Design Manual", Second Edition, Springer, 2008.
4. http://nptel.ac.in/

SUBJECT IN-CHARGE **HOD**

TRICHY MIET Mohamed Institute of Education & Technology INSTITUTIONS

M.I.E.T. ENGINEERING COLLEGE

(Approved by AICTE and Affiliated to Anna University Chennai)

TRICHY – PUDUKKOTTAI ROAD, TIRUCHIRAPPALLI – 620 007

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Sub. Code	: CS6402	**Branch/Year/Sem**	: CSE/II/IV
Sub Name	: DESIGN AND ANALYSIS OF ALGORITHMS	**Staff Name**	: P.CHRISTOPHER

COURSE OBJECTIVE

1. Learn the algorithm analysis techniques.
2. Become familiar with the different algorithm design techniques.
3. Understand the limitations of Algorithm power.

COURSE OUTCOMES

1. Interpret the fundamental needs of algorithms in problem solving.
2. Classify the different algorithm design techniques for problem solving.
3. Develop algorithms for various computing problems.
4. Analyze the time and space complexity of various algorithms.
5. Identify the limitations of algorithms in problem solving.
6. To identify the types of problem, formulate, analyze and compare the efficiency of algorithms.

Prepared by
STAFF NAME
(P.CHRISTOPHER)

Verified By
HOD

Approved by
PRINCIPAL

M.I.E.T. ENGINEERING COLLEGE
(Approved by AICTE and Affiliated to Anna University Chennai)
TRICHY – PUDUKKOTTAI ROAD, TIRUCHIRAPPALLI – 620 007

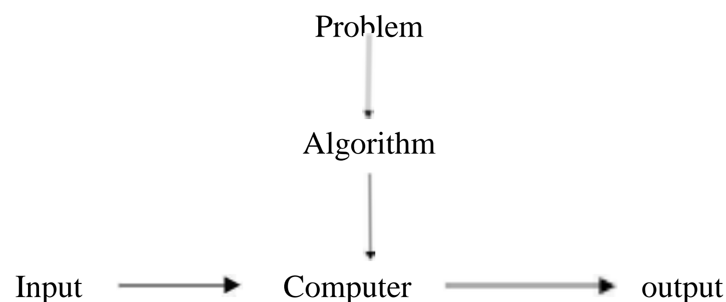
UNIT I

INTRODUCTION

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithm Efficiency – Analysis Framework – Asymptotic Notations and its properties – Mathematical analysis for Recursive and Non-recursive algorithms.

1.1 NOTION OF AN 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 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.

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

is applied repeatedly until m mod n is equal to 0;
since gcd (m, o) = 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.

1.1.1 Euclid"s algorithm :

ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n _= 0 **do**
r ←m
mod 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 mod 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: gcd (60, 24) = gcd (24,12) = 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 12 and then stop for the example given below)

1.1.2 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 pm & pn times is m and n, respectively,it should be repeated min { pm, pn } times.).
Step 4: Compute the product of all the common factors and return it as gcd of the numbers given.

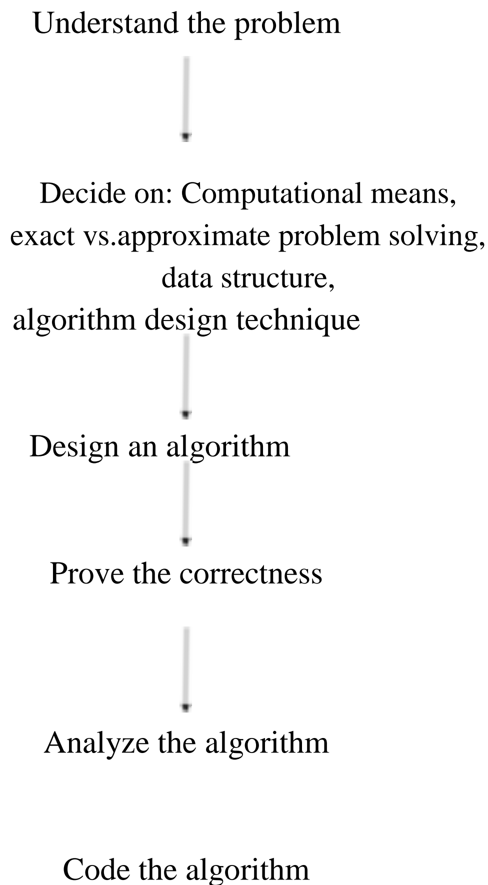
Example: 60 = 2.2.3.5
24 = 2.2.2.3

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. Code the 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 psuedocode, which is a mixture of a natural language and programming language like constructs. Its usage is similar to algorithm descriptions for writing psuedocode 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 → 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 s 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. Simper algorithms are easier to understand and program, the resulting programs will be easier to debug. For e.g. Euclid's algorithm to fid 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.

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 are 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 nlog2n 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 problems, 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. If can also lead to an accumulation of round-off errors.

The algorithms designed are mainly used in scientific and engineering applications.

1.4 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The American Heritage Dictionary defines "analysis" as the "seperation of an intellectual or substantantial 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: 0 ("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.

1.4.1 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.

1.4.1.1 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) = an x^n+ ------ + a0 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.

$$b = \log 2^{n+1}.$$

1.4.1.2 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 umber 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 Cop 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) ~ Cop 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 Cop 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) = ½ n(n-1), how much longer will the algorithm run if we doubt 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,

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

Here Cop is unknown, but still we got the result, the value is cancelled out in the ratio. Also, ½ 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.

1.4.1.3 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 nlogn 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.

1.4.1.4 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← o
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; Cworst (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 Cworst (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, Cbest(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:

1. the probability of a successful search is equal to p ($0 \le p \le 1$), and,
2. the probability of the first match occurring in the ith position is same for every i.

Accordingly, the probability of the first match occurring in the ith 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,

Cavg(n) = [1 . p/n + 2 . p/n + ………. i . p/n + ……….. n . p/n] + n.(1-p)
= p/n [1+2+…….i+……..+n] + n.(1-p)
= p/n . [n(n+1)]/2 + n.(1-p) [sum of 1st n natural number formula]

= [p(n+1)]/2 + n.(1-p)

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 tine as its input size goes to infinity.

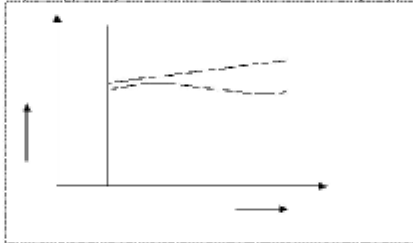
1.5 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

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



- **Ω-Notation**:

Definition: A fn t(n) is said to be in Ω[g(n)], denoted t(n) ∈Ω[g(n)], if t(n) is bounded below by some positive constant multiple of g(n) for all large n, ie., there exist some positive constant c and some non negative integer n0 s.t.

$t(n) \geq cg(n)$ for all $n \geq n0$.

For example: $n^3 \in \Omega(n^2)$, Proof is $n^3 \geq n^2$ for all $n \geq n0$. i.e., we can select c=1 and n0=0.

- θ **- Notation**:

Definition: A function t(n) is said to be in θ [g(n)], denoted t(n)∈ θ (g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, ie., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that $c2g(n) \leq t(n) \leq c1g(n)$ for all n $\geq n0$.
Example: Let us prove that $½\ n(n-1) \in \theta(n^2)$ .First, we prove the right inequality (the upper bound)

1. $n(n-1) = ½\ n^2 – ½\ n \leq ½\ n^2$ for all $n \geq n0$.

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

2. $n(n-1) = ½\ n^2 – ½\ n \geq ½\ n^2 – ½\ n½\ n$ for all $n \geq 2 = ¼$

n^2. Hence, we can select c2= ¼, c2= ½ and n0 = 2

Useful property involving these Notations:

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

THEOREMIf $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$ then $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.

PROOF (As we shall see, the proof will extend to orders of growth the following simple fact about four arbitrary real numbers a1 , b1 , a2, and b2: if $a1 < b1$ and $a2 < b2$ then $a1 + a2 < 2 \max\{b1, b2\}$.) Since $t1(n) \in O(g1(n))$, there exist some constant c and some nonnegative integer n 1 such that

$$t1(n) < c1g1(n) \text{ for all } n > n1$$

since $t2(n) \in O(g2(n))$,

$$t2(n) < c2g2(n) \text{ for all } n > n2.$$

Let us denote c3 = maxfc1, c2} and consider $n > \max\{n1, n2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$t1(n) + t2(n) < c1g1(n) + c2g2(n)$$
$$< c3g1(n) + c3g2(n) = c3\,[g1(n) + g2(n)]$$
$$< c32\max\{g1(n), g2(n)\}.$$

Hence, $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$, with the constants c and n0 required by the O definition being $2c3 = 2\max\{c1, c2\}$ and $\max\{n1, n2\}$, 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:

$$t1(n) \in O(g1(n))$$
$$t2(n) \in O(g2(n)) \text{ then } t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\}).$$

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:

t(n) 0 implies that t(n) has a smaller order of growth than g(n) lim= c implies that t(n) has the same order of growth as g(n) $n \to \infty$ g(n) ∞ implies that t (n) has a larger order of growth than g(n).

Note that the first two cases mean that t(n) Є O(g(n)), the last two mean that t(n) Є Ω(g(n)), and the second case means that t(n) Є θ(g(n)).

EXAMPLE 1 Compare orders of growth of ½ n(n - 1) and n^2. (This is one of the examples we did above to illustrate the definitions.)

$$\lim_{n \to \infty} \frac{½ n(n-1)}{n^2} = ½ \lim_{n \to \infty} \frac{n^2 - n}{n^2} = ½ \lim_{n \to \infty} (1 - 1/n) = ½$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, ½ n(n - 1) Є θ (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 10^6n^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.
logn	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
nlogn	nlogn	Many divide & conquer algorithms including mergersort quicksort fall into this class
n^2	Quadratic	Characterizes with two embedded loops, mostly sorting and matrix operations.
n3	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

1.6 MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

The general framework outlined is applied to analyze the efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates the entire 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 <- A[0]
for i <- 1 to n - 1 do
if A[i] > maxval
maxval <— 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] > maxval and the assignment maxval <- 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

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

$$\sum_{i=0}^{n} i = 1+2+\ldots+n = [n(n+1)]/2 \approx ½\, n^2 \in \theta(n^2) \quad \text{----(S2)}$$

(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 «— 0 to n — 2 do for
j' <- 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 Cworst(n) is the largest among all arrays of size n. An inspection of the innermost loop reveals that there ate two kinds of worst-case inputs (inputs for which the algorithmdoes 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

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

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

$$= (n-1)^2 - [(n-2)(n-1)]/2 = [(n-1)n]/2 \approx ½ n^2 \in \theta(n^2)$$

Also it can be solved as (by using S2)

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \ldots + 1 = [(n-1)n]/2 \approx ½ n^2 \in \theta(n^2)$$

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.

1.7 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,

$n! = 1 * 2 * \ldots * (n-1) * n = n(n-1)!$ 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!
ifn =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

$F(n) = F(n - 1) \cdot n$ for $n > 0$,

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

M(n) = M(n - 1)	+ 1	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 $M(n)$:

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

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:

M(n) = M(n - 1) + 1 substitute M(n - 1) = M(n - 2) + 1

= [M(n - 2) + 1] + 1 = M(n - 2) + 2 substitute M(n - 2) = M(n - 3) + 1

= [M(n - 3) + 1] + 2 = M (n - 3) + 3.

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:

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

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 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

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

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 [n/2] 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

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

Now backward substitutions encounter no problems:

$$A(2^k) = A(2^{k-1}) + 1 \qquad \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1$$
$$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 \text{ substitute } A(2^{k-2})$$
$$= A(2^{k-3}) + 1 = [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$$
$$\ldots\ldots\ldots$$
$$= \qquad A(2^{k-i}) + i$$
$$\ldots\ldots\ldots$$
$$= \qquad A(2^{k-k}) + k$$

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 = log2 n,

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

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 nth Fibonacci number F(n), and then we briefly discuss algorithms for computing it.

Explicit Formula for the nth 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**

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

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):

$$ar^2 + br + c = 0. \quad \text{------} \quad \mathbf{(2.6)}$$

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.

$$F(n) - F(n-1) - F(n-2) = 0. \quad \text{---------}\mathbf{(2.7)}$$

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):

$A(n) = A(n - 1) + A(n - 2) + 1$ for $n > 1$, **(2.8)**

$A(0)=0$, $A(1) = 0$.

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

$[A(n) + 1] - [A(n -1) + 1]- [A(n - 2) + 1] = 0$ and substituting $B(n) = A(n) + 1$:

$B(n) - B(n - 1) - B(n - 2) = 0$

$B(0) = 1, B(1) = 1.$

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 nth Fibonacci number

F[0]<-0; F[1]<-1

for i <- 2 to n do

F[i]«-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 nth 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 $ø^n$. If it is done by simply multiplying ø 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 nth 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 θ (logn) algorithm for computing the nth Fibonacci number that manipulates only integers. It is based on the equality

$$\begin{matrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{matrix} \quad \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix}^{n} \quad \textit{for } n \geq 1$$

and an efficient way of computing matrix powers.

UNIT II

BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force - Closest-Pair and Convex-Hull Problems-Exhaustive Search - Traveling Salesman Problem - Knapsack Problem - Assignment problem. Divide and conquer methodology – Merge sort – Quick sort – Binary search – Multiplication of Large Integers – Strassen's Matrix Multiplication-Closest-Pair and Convex-Hull Problems.

2.1 BRUTE FORCE

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.

2.1.1 CLOSEST PAIR AND CONVEX HULL PROBLEMS

2.1.1.1 CLOSEST-PAIR PROBLEM

The closest-pair problem calls for finding the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces. Points in question can represent such physical objects as airplanes or post offices as well as database records, statistical samples, DNA sequences, and so on. An air-traffic controller might be interested in two closest planes as the most probable collision candidates.Aregional postal service manager might need a solution to the closestpair problem to find candidate post-office locations to be closed.

One of the important applications of the closest-pair problem is cluster analysis in statistics. Based on n data points, hierarchical cluster analysis seeks to organize them in a hierarchy of clusters based on some similarity metric. For numerical data, this metric is usually the Euclidean distance; for

text and other nonnumerical data, metrics such as the Hamming distance are used. A bottom-up algorithm begins with each element as a separate cluster and merges them into successively larger clusters by combining the closest pair of clusters.

For simplicity, we consider the two-dimensional case of the closest-pair problem. We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i,y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$d(p_i, p_j) = (x_i - x_j)^2 + (y_i - y_j)^2$.

The brute-force approach to solving this problem leads to the following obvious algorithm: compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice. To avoid doing so, we consider only the pairs of points (p_i, p_j) for which $i < j$.

Pseudocode below computes the distance between the two closest points; getting the closest points themselves requires just a trivial modification.

ALGORITHM BruteForceClosestPair(P)
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points p1(x1, y1), . . . , pn(xn, yn)
//Output: The distance between the closest pair of
points d←∞
for i ←1 **to** n − 1 **do**
for j ←i + 1 **to** n **do**
d ←min(d, sqrt((xi
− xj)2 + (yi
− yj)2)) //sqrt is square
root **return** d

The basic operation of the algorithm is computing the square root. In the age of electronic calculators with a square-root button, one might be led to believe that computing the square root is as simple an operation as, say, addition or multiplication. Of course, it is not. For starters, even for most integers, square roots are irrational numbers that therefore can be found only approximately. Moreover, computing such approximations is not a trivial matter. But, in fact, computing square roots in the loop can be avoided! (Can you think how?) The trick is to realize that we can simply ignore the square-root function and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves.

We can do this because the smaller a number of which we take the square root, the smaller its square root, or, as mathematicians say, the square-root function is strictly increasing. Then the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows: Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant but it cannot improve its asymptotic efficiency class.

2.1.1.2 CONVEX-HULL PROBLEM

On to the other problem—that of computing the convex hull. Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important—some people believe the most important—problems in computational geometry. This prominence is due to a variety of applications in which this problem needs to be solved, either by itself or as a part of a larger task. Several such applications are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given. For example, in computer animation, replacing objects by their convex hulls speeds up collision detection; the same idea is used in path planning for Mars mission rovers.

Convex hulls are used in computing accessibility maps produced from satellite images by Geographic Information Systems. They are also used for detecting outliers by some statistical techniques. An efficient algorithm for computing a diameter of a set of points, which is the largest distance between two of the points, needs the set's convex hull to find the largest distance between two of its extreme points (see below). Finally,convex hulls are important for solving many optimization problems, because their extreme points provide a limited set of solution candidates.

A set of points (finite or infinite) in the plane is called **convex** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.All the sets depicted are convex, and so are a straight line,a triangle, a rectangle, and, more generally, any convex polygon,1 a circle, and the entire plane. On the other hand, the sets, any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex. Now we are ready for the notion of the convex hull.

Intuitively, the convex hull of a set of n points in the plane is the smallest convex polygon that contains all of them either inside or on its boundary. If this formulation does not fire up your enthusiasm, consider the problem as one of barricading n sleeping tigers by a fence of the shortest length,it is somewhat lively, however, because the fenceposts have to be erected right at the spots where some of the tigers sleep! There is another, much tamer interpretation of this notion. Imagine that the points in question are represented by nails driven into a large sheet of plywood representing the plane. Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band A formal definition of the convex hull that is applicable to arbitrary sets, including sets of points that happen to lie on the same line, follows.

The **convex hull** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.) If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart.

2.2 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.

2.2.1TRAVELING 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 vi0, vi1, …. Vin-1, vi0, 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.

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.

2.2.2 KNAPSACK PROBLEM

The problem states that: given n items of known weights w1,w 2, … wn and values v1, v2,…, vn 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, w1,w2, w3, w4 = { 7,3,4,5 } and v1,v2, v3, v4 = { 42,12,40,25 }

Subset	Total weight	Total value
Ø	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 $\Omega(2^n)$ 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.

2.2.3 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 ith person is assigned to the jth job is a known quantity
c[i,j] for each pair i,j=1,2,….,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

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

< 1,2,3,4 > cost = 9 + 4 + 1 + 4 = 18
< 1,2,4,3 > cost = 9 + 4 + 8 + 9 = 30
< 1,3,2,4 > cost = 9 + 3 + 8 + 4 = 24
< 1,3,4,2 > cost = 9 + 3 + 8 + 6 = 26 etc.

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 <j1 , j2, …., jn> in which the

i^{th} component indicates the column n of the element selected in the ith row. i.e.,The job number assigned to the ith person. For eg <2,3,4,1> 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. If requires generating all the permutations of integers 1,2,…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.

2.3 DIVIDE AND CONQUER

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 a0, a1, … an-1. If n>1, we can divide the problem into two instances: to compare 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 a0 as the solution.

i.e. a0 + a1….. + an-1 = (a0 + …..+ an/2– 1) + (an/2+ …. + 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≥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:

Є $\theta(n^d)$ where $d \geq 0$ in the above recurrence equation, then $\theta(n^d)$ if $a < b^d$
T(n) Є $\theta(n^d \log n$ if $a = b^d$ $\theta(n^{\log_b a})$
if $a > b^d$

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 $a > b^d$
A (n) Є $\theta(n^{\log_b a})$
$= \theta(n^{\log_2 2})$
$= \theta(n^1)$

2.3.1 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 el Then the elements pointed to are compared and the smaller of them is added to a new array being const smaller element is incremented to point to its immediate successor in the array it was copied from. This the two given arrays is exhausted then the remaining elements of the other array are copied to the end of the new array.

Algorithm Merge (B[0…P-1], C[0…q-1], A[0…p + q-1]) //Merge two sorted arrays into one sorted array. //Input: Arrays B[0..p-1] and C[0…q-1] both sorted
//Output: Sorted Array A [0…p+q-1] of the elements of B & C

i = 0; j = 0; k = 0 while i
< p and j < q do

if B[i] ≤ C[j]
A[k] = B[i]; i = i+1
else
A[k] = B[j]; j = j+1
K = k+1
if i=p
copy C[j..q-1] to A[k..p+q-1]
else copy B[i..p-1] to A[k..p+q-1]

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) + Cmerge (n) 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, Cmerge (n) = n-1 and the recurrence is:

Cworst (n) = 2Cworst (n/2) + n-1 for n>1, Cworst (1) = 0 When n is a power of 2, n= 2^k, by successive substitution, we get,

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) + 2Cn
= 8 C (n/8) +3 Cn
:
:

= 2^kC(1) + kCn

= an + Cnlog2n

Since k = logn and n = 2^k, we get, log2n = k(log22) = k * 1 It is easy to see that if $2^k \leq n \leq 2^{k+1}$, then

C(n) ≤ C(2^{k+1}) Є θ(nlog2n)

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: ((((4) (5) (6) ((
1 3 0

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

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

2. The stack space used for recursion. The maximum depth of the stack is proportional to log2n. 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.

2.3.2 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[o…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[o]….A[s-1] [s] A [s+1]…..A [n-1]
All are ≤ A [s] all are > 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 sub array by quick sort
//I/P: A sub array A [l..r] of A [o..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<l<r≤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 → 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 →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 ≥ 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-1 until A [j]≤ 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

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 best (n) = 2 C best (n/2) + n for n > 1 C
best (1) = 0

According to theorem, C best (n) Є θ (n log 2 n); solving it exactly for n = 2^k yields C 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…n-1] is a strictly increasing array and we use A [0] as the pivot, the L→ R scan will stop on A[1] while the R → 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:

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

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

Finally, the average case efficiency, let Cavg(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 (o ≤ s ≤ n – 1) with the same probability 1/n, we get the following recurrence relation:

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

Cavg(0) = 0 , Cavg(1) = 0

Therefore, Cavg(n) ≈ 2nln2 ≈ 1.38nlog2n

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.

2.3.3 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).

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=   0; r=             n-1
while l ≤ r do
      m=               [(l+r)/2]
      if k = A[m] return m
      else if k < A[m]  r =        m-1
      else l=      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 Cw (n) include all arrays that do not contain a search key, after one comparison the algorithm considers the half size of the array.

Cw (n) = Cw (n/2) + 1 for n > 1, Cw (1) = 1 eqn (1)

To solve such recurrence equations, assume that $n = 2^k$ to obtain the solution.
$Cw (2^k) = k + 1 = \log 2n+1$
For any positive even number n, n = 2i, where I > 0. now the LHS of eqn (1) is:

Cw (n) = [log2n]+1 = [log22i]+1 = [log 22 + log2i] + 1
= (1 + [log2i]) + 1 = [log2i] + 2

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

Cw [n/2] + 1 = Cw [2i / 2] + 1
= Cw (i) + 1
= ([log2 i] + 1) + 1
= ([log2 i] + 2

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

The worst – case efficiency is in θ (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 log2n. 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.

i.e.Cavg (n) ≈ log2n

More accurately, for successful search Cavg (n) ≈ log2n –1 and for unsuccessful search Cavg (n) ≈ log2n + 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.

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 TL and TR 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 = Ø return – 1
else return max {Height (TL), Height (TR)} + 1
```

We measure the problem's instance size by the number of nodes n(T) is 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(TL))+ A ((TR)) + 1 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 = ∅ is executed once but these 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.

The algorithm height makes one addition for every internal node of the extended treed 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:
i.e. x = n + 1

To prove this equality by induction in the no. of internal nodes n ≥ 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 thatx = K + 1 for any extended binary tree with 0 ≤ K ≤ n internal nodes. Let T be an extended binary tree with n internal nodes and x external nodes, let n L and xL be the number of internal and

external nodes in the left sub tree of T and nR & nR is the internal & external nodes of right subtree respectively. Since n > 0, T has a root which is its internal node and hence n = nL + nR + 1 using the equality

$$x = xL + xR = (nL + 1) + (nR + 1)$$
$$= (nL + nR + 1) + 1 =$$

n+1 which completes the proof

In algorithm Height, C(n), the number of comparisons to check whether the tree is empty is:
c (n) = n + x

= n + (n + 1)
= 2n + 1

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 rightsubtrees.
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.

2.3.4 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:
$23 = 2 \times 10^1 + 3 \times 10^0$ and $14 = 1 \times 10^1 + 4 \times 10^0$
Now, let us multiply them:
$23 * 14 = (2 \times 10^1 + 3 \times 10^0) * (1 \times 10^1 + 4 \times 10^0)$

$= (2*1)\, 10^2 + (3*1 + 2*4)\, 10^1 + (3*4)\, 10^0$
$= 322$

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.
i.e. $3 * 1 + 2 * 4 + (2 + 3) * (1 + 4) – (2 * 1) – (3 * 4)$
In general, for any pair of two-digit numbers a = a1 ao and b = b1 bo their product c can be computed by the formula:
$C = a * b = C2\, 10^2 + C1\, 10^1 + Co$
Where,
C2 = a1 * b1 – Product of 1st digits
Co = ao * bo – Product of their 2nd digits
a1 + ao) * (b1 + bo) – (C2 + Co) product of the sum of the a's digits and the sum of the b's digits minus the sum of C2 and Co.

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 a1 and ao respectively. Similarly, for b it is b1 and bo respectively.
Here, a = a1 ao implies that $a = a1\, 10^{n/2} + ao$

And, b = b1 bo implies that b = b1 $10^{n/2}$ + bo

Therefore, C = a * b = (a1 $10^{n/2}$ + ao) * (b1 $10^{n/2}$ + bo)

= (a1 * b1) 10^{n} + (a1* b0 + ao * b1) $10^{n/2}$ + (a0* b0)

= C2$10^{n}$ + C1$10^{n/2}$ + Co

Where, C2 = a1 * b1 – Product of first half Co =

ao * bo – Product of second half

a1 + ao) * (b1 + bo) – (C2 + Co) product of the sum of the a's halves and the sum of the b's halves minus the sum of C2 & Co.

If n/2 is even, we can apply the same method for computing the products C2, Co, and C1. 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) for n>1, M(1) = 1

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

$M(2^K) = 3 M(2^{K-1}) = 3 [3M(2^{K-2})] = 3^2 M(2^{K-2})$

$= \ldots 3^i M(2^{k-i}) = \ldots 3^K M(2^{K-K}) = 3^K$

Since k = log2n,

$M(n) = 3^{\log_2 n}$

$= n^{\log_2 3} \approx n^{1.585}$

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

2.3.5 STRASSEN'S MATRIX MULTIPLICATION

This is a problem which is used for multiplying two n x 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.

a00 * b00 + a01 * b10	a00 * b01 + a01 * b11
a10 * b00 + a11 * b10	a10 * b01 + a11 * b11

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

$$\begin{bmatrix} m1 + m4 - m5 + m7 & m3 + m5 \\ m2 + m4 & m1 + m3 - m2 + m6 \end{bmatrix}$$

Where,
m1 = (a00 + a11) * (b00 + b11)
m2 = (a10 + a11)* b00
m3 = a00 * (b01 - b11)
m4 = a11 * (b10 – b00)
m5 = (a00 + a01 * b11
m6 = (a10 – a00) * (b00 + b01)
m7 = (a01 - a11) * (b10 + b11)

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 matrices as follows:

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:

$M(n) = 7M (n/2)$ for $n>1$, $M(1) = 1$

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

$M (2^K) = 7 M (2^{K-1}) = 7 [7M (2^{K-2})] = 7^2 M(2^{K-2})$

$= \dots 7^i M (2^{k-i}) = \dots 7^K M (2^{K-K}) = 7^K$

Since $k = \log 2n$,
$M(n) = 7^{\log_2 n}$
$= n^{\log_2 7}$
$\approx n^{2.807}$

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 matrices of order n/2 and make 18 additions of matrices of size n/2; when n=1, no additions are made since two numbers are simply multiplied. The recurrence relation is

$A(n) = 7\,A(n/2) + 18\,(n/2)^2$ for n>1

$A(1) = 0$

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.

2.3.6 THE CLOSEST-PAIR AND CONVEX-HULL PROBLEMS BY DIVIDE-AND-CONQUER

2.3.6.1 THE CLOSEST-PAIR PROBLEM

Let P be a set of n > 1 points in the Cartesian plane. For the sake of simplicity, we assume that the points are distinct. We can also assume that the points are ordered in nondecreasing order of their x coordinate. (If they were not, we could sort them first by an efficeint sorting algorithm such as mergesort.) It will also be convenient to have the points sorted in a separate list in nondecreasing order of the y coordinate; we will denote such a list Q.

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If n > 3, we can divide the points into two subsets Pl and Pr of _n/2_ and _n/2_points, respectively, by drawing a vertical line through the median m of their x coordinates so that _n/2_ points lie to the left of or on the line itself, and _n/2_ points lie to the right of or on the line. Then we can solve the closest-pair problem recursively for subsets Pl and Pr . Let dl and dr be the smallest distances between pairs of points in Pl and Pr , respectively, and let d = min{dl, dr}.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2d around the separating line, since the distance between any other pair of points is at least d obtained from Q and hence ordered in nondecreasing order of their y coordinate.

Scan this list, updating the information about dmin, the minimum distance seen so far, if we encounter a closer pair of points. Initially, dmin = d, and subsequently dmin ≤ d. Let p(x, y) be a point on this list. For a point p(x, y) to have a chance to be closer to p than dmin, the point must follow p on list S and the difference between their y coordinates must be less than dmin.

Geometrically,this means that p must belong to the rectangle. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart. It is easy to prove that the total number of such points in the rectangle, including p, does not exceed eight a more

careful analysis reduces this number to six .Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point. Here is pseudocode of the algorithm.

ALGORITHM EfficientClosestPair(P, Q)
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
// nondecreasing order of their x coordinates and an array Q of the
// same points sorted in nondecreasing order of the y coordinates //Output: Euclidean distance between the closest pair of points **if** n ≤ 3
return the minimal distance found by the brute-force algorithm **else**
copy the first _n/2_ points of P to array Pl copy the same _n/2_ points from Q to array Ql copy the remaining _n/2_ points of P to array Pr copy the same _n/2_ points from Q to array Qr dl

←EfficientClosestPair(Pl, Ql) dr ←EfficientClosestPair(Pr, Qr) d ←min{dl, dr

}
m←P[_n/2_ − 1].x
copy all the points of Q for which |x − m| < d into array S[0..num − 1] dminsq ←d2
for i ←0 **to** num − 2 **do** k←i + 1
while k ≤ num − 1 **and** (S[k].y − S[i].y)2 < dminsq
dminsq ←min((S[k].x − S[i].x)2+ (S[k].y − S[i].y)2, dminsq) k←k + 1
return sqrt(dminsq)

the following recurrence for the running time of the algorithm:
T (n) = 2T (n/2) + f (n), where f (n) ∈ _(n).

Applying the Master Theorem (with a = 2, b = 2, and d = 1), we get T (n) ∈ _(n log n). The necessity to presort input points does not change the overall efficiency class if sorting is done by a O(n log n) algorithm such as mergesort. In fact, this is the best efficiency class one can achieve, because it has been proved that any algorithm for this problem must be in _(n log n) under some natural assumptions about operations an algorithm can perform.

2.3.6.2 CONVEX-HULL PROBLEM

Let us revisit the convex-hull problem,: find the smallest convex polygon that contains n given points in the plane. We consider here a divide-and-conquer algorithm called **quickhull** because of its resemblance to quicksort.

Let S be a set ofn>1 points p1(x1, y1), . . . , pn(xn, yn) in the Cartesian plane.We assume that the points are sorted in nondecreasing order of their x coordinates,with ties resolved by increasing order of the y coordinates of the points involved.It is not difficult to prove the geometrically obvious fact that the leftmost point p1 and the rightmost point pn are two distinct extreme points of the set's convex hull

Let −−−→p1pn be the straight line through points p1 and pn directed from p1 to pn. This line separates the points of S into two sets: S1 is the set of points to the left of this line, and S2 is the set of points to the right of this line. (We say that point q3 is to the left of the line −−−−→ q1q2 directed from point q1 to point q2 if q1q2q3 forms a counterclockwise cycle. Later, we cite an analytical way to check this condition, based on checking the sign of a determinant formed by the coordinates of the three points.) The points of S on the line −−−→p1pn, other than p1and pn, cannot be extreme points of the convex hull and hence are excluded from further consideration.

The boundary of the convex hull of S is made up of two polygonal chains:an "upper" boundary and a "lower" boundary. The "upper" boundary, called the **upper hull**, is a sequence of line segments with vertices at p1, some of the points in S1 (if S1 is not empty) and pn. The "lower" boundary, called the **lower hull**, is a sequence of line segments with vertices at p1, some of the points in S2 (if S2 is not empty) and pn. The fact that the convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation exploited by several algorithms for this problem.

For concreteness, let us discuss how quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If S1 is empty, the empty, the algorithm identifies point pmax in S1, which is the farthest from the line −−−→ p1pn . If there is a tie, the point that maximizes the angle _ pmaxppn can be selected. (Note that point pmax maximizes the area of the triangle with two vertices at p1 and pn and the third one at some other point of S1.) Then the algorithm identifies all the points of set S1 that are to the left of the line −−−→ p1pmax;these are the points that will make up the set S1,1. The points of S1 to the left of the line −−−−−−→ pmaxpn will make up the set S1,2. It is not difficult to prove the following: pmax is a vertex of the upper hull.

The points inside _p1pmaxpn cannot be vertices of the upper hull (and hence can be eliminated from further consideration).There are no points to the left of both lines −−−→p1pmax and −−−−−−→ pmaxpn. Therefore, the algorithm can continue constructing the upper hulls of p1 ∪ S1,1 ∪ pmax and pmax ∪ S1,2 ∪ pn recursively and then simply concatenate them to get the upper hull of the entire set p1 ∪ S1 ∪ pn. actually implemented. Fortunately, we can take advantage of the following very useful fact from analytical geometry: if q1(x1, y1), q2(x2, y2), and q3(x3, y3) are three arbitrary points in the Cartesian plane, then the area of the triangle _q1q2q3 is equal to one-half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3,$$

while the sign of this expression is positive if and only if the point q3 = (x3, y3) is to the left of the line −−−→ q1q2. Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

UNIT III

DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

Computing a Binomial Coefficient – Warshall's and Floyd' algorithm – Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique– Prim's algorithm- Kruskal's Algorithm- Dijkstra's Algorithm-Huffman Trees.

3.1 DYNAMIC ROGRAMMING

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 arises 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.

3.1.1 COMPUTING A BINOMIAL COEFFICIENT

It's used to apply in nonoptimization problem. Elementary combinatorics, 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:
$(a+b)^n = c(n,0)a^n + \ldots + c(n,i)a^{n-i}b^i + \ldots + c(n,n)b^n$
$c(n,k) = c(n-1,k-1) + c(n-1,k)$ for $n>k>0$ and $c(n,0)= c(n,n) = 1$

	0	1	………………………………k-1	
0	1			
1	1	1		
2	1	2		
.				
.				
.				
k	1			
.				
.				
.				
n-1	1		c(n-1,k-1)	c(n-1,k)
n	1			(n,k)

```
   Algorithm Binomial (n,k)
// Computes c(n,k)
// Input:A pair of non –ive int n ≥ k ≥ 0
// Output:The value of c(n,k)
   for i 0 to n do
   for j 0 to min(i,k) do if
   j=0 or j=k
   c[i,j]  1
   else c[i,j] 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

	1				n
(n,k) = ∑	1 + ∑∑ 1 = ∑ (i-1) + ∑				k
=		=k			i=k
	=1	1	=1	=1	+1

= [(k-1)k]/2 + k(n-k) Є θ(nk)

3.2 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.

3.2.1 Warshall's algorithm:

The adjacency matrix A={aij} of a directed graph is the Boolean matrix that has 1 in its ith row and j^{th} column iff there is a directed edge from the ith vertex to j^{th} vertex.

Definition:- The transitive closure of a directed graph with n vertices can be defined as the n x n Boolean matrix T={tij}, in which the element in the ith row (1<=i<=n) and the jth column ($1 \le j \le n$) is 1 if there exists a nontrivial directed path (i.e., directed path of positive length) from the ith vertex to the jth vertex; otherwise ,tij is 0.

We can generate it with DFS or BFS .It constructs the transitive closure of a given diagraph with n vertices through a series of n x n matrices.

R0, R (1), ... R (k-1) , R (k),......R(n) .

The element $rij^{(k)}$ in the ith row and j^{th} col of matrix $R^{(k)}$ = 1 iff there exists a directed path from the ith 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 1st 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 ith row and j^{th} col of matrix $R^{(k)}$,be equal to 1.This means that there exists a path from i^{th} vertex vi to the jth vertex vj with each intermediate vertex numbered not higher than k.

vi, a list of intermediate vertices each numbered not higher than k,vj.

There are 2 possibilities. First is, if there is a list the intermediate vertex doesnot contains the kth vertex. The path from vi to vj has vertices not higher than k-1, and therefore $rij^{(k-1)}=1$. The second is, that path does contain the kth vertex vk among the intermediate vertices. vk occurs only once in the list. Therefore vi,vertices numbered ≤k-1, vk, vertices numbered ≤k-1, vj.
Means there exists a path from vi to vk with each intermediate vertex numbered not higher than k- 1 hence $rik^{(k-1)}=1$ and the 2nd part is such that a path from vk to vj with each intermediate vertex numbered not higher than
k-1 hence, $rkj^{(k-1)} = 1$ i.e., if $rij^{(k)} = 1$,then either $rij^{(k-1)} = 1$ or both $rik^{(k-1)}=1$ and $rkj^{(k-1)}=1$.

Thus the formula is : $rij^{(k)} = rij^{(k-1)}$ or $rik^{(k-1)}$ and $rkj^{(k-1)}$ This formula yields the Warshall's algorithm which implies:

- If an element rij is 1 in $R^{(k-1)}$,it remains 1 in $R^{(k)}$.
- If an elt rij 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 $R^{(k-1)}$.

```
Algorithm Warshall (A[1..n,1..n])
R(0)    A
for k 1 to n do for i 1
to n do
for j 1 to n do
R(k)[i,j] R R(k-1) [i,j] or R R(k-1) [i,k] and R R(k-1) [k,j]
Return R(n) .
```

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.

3.2.1 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 x n matrix D called distance matrix. The element dij in the ith row and j^{th} column of this matrix indicates the length of the shortest path from the ith vertex to j^{th} vertex ($1 \leq i,j \leq n$).

1).

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 x n matrices.
D0, D (1), …… D (k-1) , D (k),…… D(n)

$D^{(0)}$ with no intermediate vertices and $D^{(n)}$ consists all the vertices as intermediate vertices. $dij^{(k)}$ is the shortest path from vi to vj with their intermediate vertices numbered not higher than k. $D^{(k)}$ is obtained from its immediate predecessor $D^{(k-1)}$.

i.e., vi a list of int. vertices each numbered not higher than k, vj.

We can partition the path into 2 disjoint subsets: those that do not use the kth 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 dij

The length of the shortest path in the 2nd subset is,if the graph doesnot contain a cycle of negative length, the subset use vertex vk as intermediate vertex exactly once.

vi,vertices numbered $\leq$ k-1, vk, vertices numbered $\leq$ k-1, vj .

The path vi to vk is equal to $dik^{(k-1)}$ and vk to vij is $dkj^{(k-1)}$,the length of the shortest path among the paths that use the kth vertex is equal to $dik^{(k-1)} + dkj^{(k-1)}$.

$dij^{(k)} = \min\{ dij^{(k-1)}, dik^{(k-1)} + dkj^{(k-1)} \}$ for $k \geq 1$,
$dij^{(0)} = wij$.

```
Algorithm Floyd(w[1..n,1..n])
//Implemnt 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 w
For k 1 to n do
```

For i 1 to n do
For j 1 to n do
D[i,j] min {D[i,j], D[i,k] + D[k,j]}

Return D.

The time efficiency is θ (n^3).The principle of optimality holds for these problems where there is an optimization.

3.3 OPTIMAL BINARY SEARCH TREES

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion. If probabilities of searching for elements of a set are known—e.g., from accumulated data about past searches—it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible. For simplicity, we limit our discussion to minimizing the average number of comparisons in a successful search. The method can be extended to include unsuccessful searches as well.As an example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. The average number of comparisons in a successful search in the first of these trees is 0.1 . 1+ 0.2 . 2 + 0.4 .3+ 0.3 . 4 = 2.9, and for the second one it is 0.1 . 2
+ 0.2 . 1+ 0.4 . 2 + 0.3 . 3= 2.1.Neither of these two trees is, in fact, optimal. (Can you tell which binary tree is optimal?) For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys. As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with n keys is equal to the nth **Catalan number**,

c(n) = 1/(n + 1)(2n/n) for n > 0, c(0) = 1,

which grows to infinity as fast as 4n/n1.5 (see Problem 7 in this section's exercises). So let a1, . . . , an be distinct keys ordered from the smallest to the largest and let p1, . . . , pn be the probabilities of searching for them. Let C(i, j) be the smallest average number of comparisons made in a successful search in a binary search tree Tji made up of keys ai, . .
. , aj , where i, j are some integer indices, $1 \le i \le j \le n$.Following the classic dynamic programming approach, we will find values of C(i, j) for all smaller instances of the problem, although we are interested just in C(1, n). To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root ak among the keys ai, . . . , aj . For such a binary search tree (Figure 8.8), the root contains key ak, the left subtree T k−1 i contains keys ai, . . . , ak−1 optimally arranged, and the right subtree Tjk+1 contains keys ak+1, . . . , aj also optimally arranged. (Note how we are taking advantage of the principle of optimality here.).

The two-dimensional table has the values needed for computing C(i, j) by formula ,they are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of C(i, j). This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities pi, $1 \le i \le n$, right above it and moving toward the upper right corner.

The algorithm we just sketched computes C(1, n)—the average number of comparisons for successful searches in the optimal binary tree. If we also want to get the optimal tree itself, we need to maintain another two-dimensional table to record the value of k for which the minimum is achieved. The table has the same shape as the table in and is filled in the same manner, starting with entries R(i, i) = i for $1 \le i \le n$. When the table is filled, its entries indicate indices of the roots of the optimal subtrees, which makes it possible to reconstruct an optimal tree for the entire set given.

EXAMPLE Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

C(i, j) = mini≤k≤j {C(i, k − 1) + C(k + 1, j)} +s=ito j ps for 1≤ i ≤ j ≤ n.

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

ALGORITHM OptimalBST(P [1..n])

//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
// optimal BST and table R of subtrees' roots in the optimal BST

for i ←1 **to** n **do**
C[i, i − 1]←0
C[i, i]←P[i]
R[i, i]←i
C[n + 1, n]←0
for d ←1 **to** n − 1 **do** //diagonal count
for i ←1 **to** n − d **do**
j ←i + d
minval←∞
for k←i **to** j **do**
if C[i, k − 1]+ C[k + 1, j]< minval
minval←C[i, k − 1]+ C[k + 1, j]; kmin←k
R[i, j]←kmin
sum←P[i]; **for** s ←i + 1 **to** j **do** sum←sum + P[s] C[i,
j]←minval + sum
return C[1, n], R

The algorithm's space efficiency is clearly quadratic; the time efficiency of this version of the algorithm is cubic (why?).Amore careful analysis shows that entries in the root table are always nondecreasing along each row and column. This limits values for R(i, j) to the range R(i, j − 1), . . . , R(i + 1, j) and makes it possible to reduce the running time of the algorithm to _(n2).

3.4 THE KNAPSACK PROBLEM AND MEMORY FUNCTIONS

The knapsack problem states that given n items of known weights w1, w2 ,…, wn and values v1, v2 ,….., vn 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 ith item and those that do.

1. Among the subsets that do not include the ith item, the value of an optimal subset is,by definition: v[i-1,j]

2. Among the subsets that do not include the ith 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-wi. The value of such an optimal subset is vi+v[i-1,j-wi].

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 ith 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.

Therefore, v[i,j] = { max{ v[i-1,j], vi + v[i-1,j-wi] },if j-wi ≥ 0 v[i-1,j], if j-wi< 0.
The initial conditions are:
v[0,j] = 0 for j ≥ 0 and v[i,0] = 0 for i ≥ 0.
Our goal is to find v[n,w],the maximal value of a subset of n items which fit into W.

The maximal value is v[4,5]=$37. Since v[4,5] ≠ 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] ≠ 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] ≠ 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. Intially, 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.

```
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< weights[i]
value MF knapsack(i-1, j) else
  value max (MF knapsack(i-1,j),values[i]+MF knapsack(i-1,j-weights[i]) v[i,j] value
return
v[i,j].
```

3.5 GREEDY TECHNIQUE

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 d1 >d2……dm. For example: d1
=25, d1=10, d3=5, d4=1. To get a change for 48 cents. First step is to give 1 d1, 2 d2 and 3 d4'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.

3.5.1 PRIM'S ALGORITHM

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 ; T1 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: ET, the set of edges composing a minimum spanning tree of G.
VT <-{V0}
ET <- ø
For i <- 1 to |V|-1 do
Find a minimum weight edge e*=(v*,u*) among all the edges (v,u)
such that v is in VT & u is in V-VT.

VT <- VT U
{u*} ET <- ET
U {e*}

Return ET.

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-VT to the set of tree vertices VT.

- For each remaining vertex u in V-VT 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 T0 consists of a single vertex and hence must be a part of any Minimum Spanning Tree. Assume that Ti-1 is part of some Minimum Spanning Tree. We need to prove that Ti generated from Ti-1 is also a part of Minimum Spanning Tree, by contradiction. Let us assume that no Minimum Spanning Tree of the graph can contain Ti. let ei = (v,u) be the minimum weight edge from a vertex in Ti-1 to a vertex not in Ti-1 used by Prim's algorithm to expand T i-1 to Ti, e i cannot belong to any Minimum Spanning Tree including T. Therefore if we add ei to T, a cycle must be formed.

In addition to edge ei = (v,u), this cycle must contain another edge (v',u') connecting a vertex v' Є Ti-1 to a vertex u' which is not in Ti-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 ei 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 Ti.

The efficieny of this algorithm depends on the data structure chosen for the graph itself and for the p vertex priorities are the distance to the nearest tree vertices. For eg, if the graph is represented by weigh 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(logn) 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 ,$|V|-1 \leq |E|$.

3.5.2 KRUSKAL'S ALGORITHM

It looks at Minimum Spanning Tree for a weighted connected graph G=<V,E> 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=<V,E>
//  Output: ET,the set of edges composing a Minimum Spanning
Tree of G Sort E in increasing order of edge weights
ET<- ø ; ecounter <- 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.
```

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 S1,S2,S3….. Sk. 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.

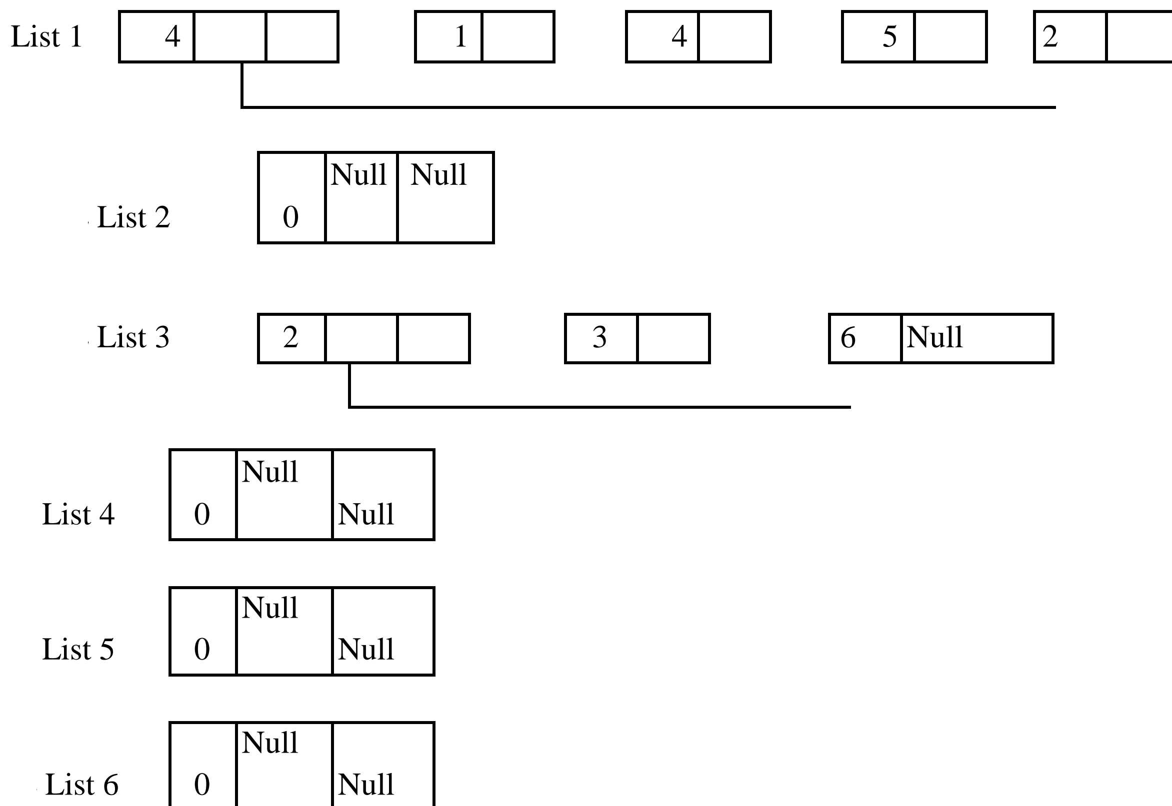
- Union(x,y): Constructs the union of the disjoint subsets Sx & Sy containing x & y respectively and adds it to the collection to replace Sx & Sy, which are deleted from it.

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.

Size Last First



Subset representatives

Element Index	Representation
1	1
2	1
3	3
4	1
5	1
6	3

For the makeset(x) time is in θ(1) and for 'n' elements it will be in θ(n). The efficiency of find(x) is also in θ (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 θ (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 θ(n) time. Therefore, the total time needed will be θ(nlog n).

Each ai is updated Ai times, the resulting set will have 2^{Ai} elements. Since the entire set has n elements, $2^{Ai} \le n$ and hence A i ≤ log2n. Therefore, the total number of possible updates of the representatives for all n elements is S will not exceed nlog2n.

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 θ (1) time and for 'n' elements it is θ (n), a union (x,y) is θ (1) and find(x) is in θ (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 he execution of a find operation point to the tree's root.

3.5.3 DIJKSTRA'S ALGORITHM

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 is considered. It is a set pf 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 ith iteration commences, the algorithm has already, identified its shortest path to i -1 other vertices nearest to the source. This forms a sub tree Ti and the next vertex chosen to be should be vertices adjacent to the vertices of Ti, fringe vertices. To identify , the ith 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. du* + w(u*,u) < du, update the labels of u by u* and , du* + w(u*,u) respectively.

Algorithm Dijkstra(a)
// Input: A weighted connected graph G=<V,E> and its vertex s
// Output: The length dv of a shortest path from s to v and its penultimate vertex pv //
// 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
dv <- ∞; pv <-null
Insert(Q,v, dv) //initialize vertex priority in the priority queue ds
<- 0; decrease(Q,s, ds) //update priority of s with ds VT <- ø
for i<-0 to |V|-1 do
u* <- deleteMin(Q) //delete the minimum priority element
VT <- VT U {u*}
for every vertex u in V-VT that is adjacent to u* do
if du* + w(u*,u) < du
du <- du* + w(u*,u); pu <-
u* Decrease(Q,u, du)

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|).

3.6 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 sumof 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	__
robability	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.

Huffman's code is not only limited to data compression. The sum ∑ liwi where i=1
li 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.

UNIT IV

ITERATIVE IMPROVEMENT

The Simplex Method-The Maximum-Flow Problem – Maximm Matching in Bipartite Graphs-
The Stable marriage Problem.

4.1 THE SIMPLEX METHOD

Variables u and v, transforming inequality constraints into equality constrains, are called slack variables

A basic solution to a system of m linear equations in n unknowns ($n \geq m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other m variables. The variables set to 0 are called nonbasic; the variables obtained by solving the system are called basic.
A basic solution is called feasible if all its (basic) variables are nonnegative.

Example $x + y + u = 4$

$x + 3y + v = 6$

(0, 0, 4, 6) is basic feasible solution

(x, y are nonbasic; u, v are basic)

There is a 1-1 correspondence between extreme points of LP's feasible region
and its basic feasible solutions.

Maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0, y \geq 0, u \geq 0, v \geq 0$

basic feasible solution

(0, 0, 4, 6)

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	−3	−5	0	0	0

basic variables (u, v); objective row (last row); value (0); basic f

Simplex method:

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau. Step 1 [Optimality test] If all entries in the objective row are nonnegative — stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select (the most) negative entry in the objective row. Mark its column to indicate the entering variable and the pivot column.

Step 3 [Find departing variable] For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — stop: the problem is unbounded.) Find the row with the smallest θ-ratio, mark this row to indicate the departing variable and the pivot row.

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

imize $z = 3x + 5y + 0u + 0v$

ect to $y + u = 4$

$3y = 6$

$y \geq 0, u \geq 0,$

4.2 THE MAXIMUM-FLOW PROBLEM

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- contains exactly one vertex with no entering edges, called the source (numbered 1)
- contains exactly one vertex with no leaving edges, called the sink (numbered n)
- has positive integer weight uij on each directed edge (i.j), called the edge capacity, indicating the upper bound on the amount of the material that can be sent from i to j through this edge

Definition of flow:

A flow is an assignment of real numbers xij to edges (i,j) of a given network that satisfy the following:

flow-conservation requirements: The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

capacity constraints

$0 \leq x_{ij} \leq u_{ij}$ for every edge (i,j) □ E **Flow**

value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

j xjn

j) є E n) є E

The value of the flow is defined as the total outflow from the source (= the total inflow into the sink).

Augmenting Path (Ford-Fulkerson) Method

Start with the zero flow (xij = 0 for every edge)

- On each iteration, try to find a flow-augmenting path from source to sink, which a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum

Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i,j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$

– known as forward edge (→)

OR

- connected by a directed edge (j to i) with positive flow x_{ji}

– known as backward edge (←)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

$r = \min\{r_{ij}$ on all forward edges, x_{ji} on all backward edges$\}$

- Assuming the edge capacities are integers, r is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's Efficiency.

Shortest-Augmenting-Path Algorithm

Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled

second label – indicates the vertex from which the vertex being labeled was reached, with "+" or "–" added to the second label to indicate whether the vertex was reached via a forward or backward edge

The source is always labeled with ∞,-

All other vertices are labeled as follows:

If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (forward edge), vertex j is labeled with $l_j, i+$, where $l_j = \min\{l_i, r_{ij}\}$

If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled $l_j, i-$, where $l_j = \min\{l_i, x_{ji}\}$

If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label

The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path

If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops

Definition of a Cut:

Let X be a set of vertices in a network that includes its source but does not include its sink, and let X, the complement of X, be the rest of the vertices including the sink. The cut induced by this partition of the vertices is the set of all the edges with a tail in X and a head in X.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

We'll denote a cut and its capacity by C(X,X) and c(X,X)

Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink

Minimum cut is a cut of the smallest capacity in a given network

Max-Flow Min-Cut Theorem

The value of maximum flow in a network is equal to the capacity of its minimum cut.

The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:

- o maximum flow is the final flow produced by the algorithm
- o minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices

on the last iteration of the algorithm all the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

Time Efficiency

The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds nm/2, where n and m are the number of vertices and edges, respectively

Since the time required to find shortest augmenting path by breadth-first search is in O(n+m)=O(m) for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in O(nm2) for this representation

More efficient algorithms have been found that can run in close to O(nm) time, but these algorithms don't fall into the iterative-improvement paradigm.

4.3 MAXIMUM MATCHING IN BIPARTITE GRAPHS

Bipartite Graphs

Bipartite graph: a graph whose vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U.

A graph is bipartite if and only if it does not have a cycle of an odd length

A bipartite graph is 2-colorable: the vertices can be colored in two colors so that every edge has its vertices colored differently

Matching in a Graph

A matching in a graph is a subset of its edges with the property that no two edges share a vertex a matching in this graph.

A maximum (or maximum cardinality) matching is a matching with the largest number of edges

1) always exists

2) not always unique

Free Vertices and Maximum Matching

A matching in this graph (M)

For a given matching M, a vertex is called free (or unmatched) if it is not an endpoint of any edge in M; otherwise, a vertex is said to be matched

If every vertex is matched, then M is a maximum matching

If there are unmatched or free vertices, then M may be able to be improved

We can immediately increase a matching by adding an edge connecting two free vertices

Augmenting Paths and Augmentation

An augmenting path for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M

The length of an augmenting path is always odd

Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (augmentation) One-edge path between two free vertices is special case of augmenting path

Matching on the right is maximum (perfect matching)

Theorem A matching M is maximum if and only if there exists no augmenting path with respect to M

Augmenting Path Method (template)

Start with some initial matching. e.g., the empty set

Find an augmenting path and augment the current matching along that path o e.g., using breadth-first search like method

When no augmenting path can be found, terminate and return the last matching, which is maximum?

BFS-based Augmenting Path Algorithm

Initialize queue Q with all free vertices in one of the sets (say V)

While Q is not empty, delete front vertex w and label every unlabeled vertex u adjacent to w as follows:

Case 1 (w is in V): If u is free, augment the matching along the path ending at u by moving backwards until a free vertex in V is reached. After that, erase all labels and reinitialize Q with all the vertices in V that are still free

If u is matched (not with w), label u with w and enqueue u

Case 2 (w is in U) Label its matching mate v with w and enqueue v

After Q becomes empty, return the last matching, which is maximum

Each vertex is labeled with the vertex it was reached from. Queue deletions are indicated by arrows. The free vertex found in U is shaded and labeled for clarity; the new matching obtained by the augmentation is shown on the next slide.

This matching is maximum since there are no remaining free vertices in V (the queue is empty)

Note that this matching differs from the maximum matching found earlier.

4.4 THE STABLE MARRIAGE PROBLEM

Stable Marriage Problem

There is a set Y = {m1,…,mn} of n men and a set X = {w1,…,wn} of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).

A marriage matching M is a set of n pairs (mi, wj).

A pair (m, w) is said to be a blocking pair for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.

A marriage matching M is called stable if there is no blocking pair for it; otherwise, it’s called unstable.

The stable marriage problem is to find a stable marriage matching for men’s and women’s given preferences.

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:
Proposal The selected free man m proposes to w, the next woman on his preference list

Response if w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her current mate. If she prefers m to him, she accepts m's proposal, making her former mate free; otherwise, she simply rejects m's proposal, leaving m free
Step 2 Return the set of n matched pairs

The algorithm terminates after no more than n2 iterations with a stable marriage output

The stable matching produced by the algorithm is always man-optimal: each man gets the highest rank woman on his list under any stable marriage. One can obtain the woman-optimal matching by making women propose to men

A man (woman) optimal matching is unique for a given set of participant preferences
The stable marriage problem has practical applications such as Matching medical-school graduates with hospitals for residency training.

UNIT V

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

Limitations of Algorithm Power-Lower-Bound Arguments-Decision Trees-P, NP and NP-Complete Problems--Coping with the Limitations - Backtracking – n-Queens problem – Hamiltonian Circuit Problem – Subset Sum Problem-Branch and Bound – Assignment problem – Knapsack Problem – Traveling Salesman Problem- Approximation Algorithms for NP – Hard Problems – Traveling Salesman problem – Knapsack problem.

5.1 LIMITATIONS OF ALGORITHM POWER

A fair assessment of algorithms as problem-solving tools is inescapable: they are very powerful instruments, especially when they are executed by modern computers. But the power of algorithms is not unlimited, and its limits are the subject of this chapter. As we shall see, some problems cannot be solved by any algorithm. Other problems can be solved algorithmically but not in polynomial time. And even when a problem can be solved in polynomial time by some algorithms, there are usually lower bounds on their efficiency.

5.1.1 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.

5.1.2 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.

Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs.We can study the performance of such algorithms with a device called a **decision tree**. As an example, a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., k < k . The node's left subtree contains the information about subsequent comparisons made if k < k , and its right subtree does the same for the case of k >k. (For the sake of implicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm's run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons. An important point is that 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 length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree.

The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves. Specifically, it is not difficult to prove that for any binary tree with l leaves and height h, h ≥ _log2 l_) Indeed, a binary tree of height h with the largest number of leaves has all its leaves on the last level (why?). Hence, the largest number of leaves in such a tree is 2h. In other words, 2h ≥ l, which immediately implies. Inequality puts a lower bound on the heights of binary decision trees and hence the worst-case number of comparisons made by any comparison-based algorithm for the problem in question. Such a bound is called the

Information theoretic lower bound.

We illustrate this technique below on two important problems: sorting and searching in a sorted array.

Decision Trees for Searching a Sorted Array

We shall see how decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: A[0]<A[1]< . . .<A[n − 1]. The principal algorithm for this problem is binary search. The number of comparisons made by binary search in the worst case, Cbsworst(n), is given by the formula Cbsworst(n) = _log2 n_ + 1= _log2(n + 1)_.

Since we are dealing here with three-way comparisons in which search keyK is compared with some element A[i] to see whetherK <A[i],K = A[i], orK >A[i], it is natural to try using ternary decision trees. Figure 11.4 presents such a tree for the case of n = 4.The internal nodes of that tree indicate the array's elements being compared with the search key. The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search.

For an array of n elements, all such decision trees will have 2n + 1 leaves (n for successful searches and n + 1 for unsuccessful ones). Since the minimum height h of a ternary tree with l leaves is _log3 l_, we get the following lower bound on the number of worst-case comparisons:

Cworst(n) ≥ _log3(2n + 1).

This lower bound is smaller than _log2(n + 1)_, the number of worst-case comparisons for binary search, at least for large values of n (and smaller than or equal to _log2(n + 1)_ for every positive integer n—see Problem 7 in this section's exercises). Can we prove a better lower bound, or is binary search far from being optimal? The answer turns out to be the former. To

obtain a better lower bound, we should consider binary rather than ternary decision trees, such as the one in Figure 11.5. Internal nodes in such a tree correspond to the same threeway comparisons as before, but they also serve as terminal nodes for successful searches. Leaves therefore represent only unsuccessful searches, and there are n + 1 of them for searching an n-element array.

As comparison of the decision trees in, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality to such binary decision trees immediately yields Cworst(n) ≥ _log2(n + 1)**)**. This inequality closes the gap between the lower bound and the number of worstcase comparisons made by binary search, which is also _log2(n + 1)_. A much more sophisticated analysis shows that under the standard assumptions about searches, binary search makes the smallest number of comparisons on the average, as well. The average number of comparisons made by this algorithm turns out to be about log2 n − 1 and log2(n + 1) for successful and unsuccessful searches, respectively.

5.1.3 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≤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 D2 if there exists a function t that transforms instances of D1 to instances of D2 such that

1) t maps all yes instances of D1 to yes instances of D2 and all no instances of D1 to no instances of D2;
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: (x1 v x2 v x3) & (x1 v x2)

Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called **polynomial**.

It is natural to wonder whether every decision problem can be solved in polynomial time. The answer to this question turns out to be no. In fact, some decision problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm. A famous example of an undecidable problem was given by Alan Turing in 1936. The problem in question is called the **halting problem**: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

NP-Complete Problems

Informally, an 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 Here are more formal definitions of these concepts.

decision problem D1 is said to be **polynomially reducible** to a decision problem D2, if there exists a function t that transforms instances of D1 to instances of D2 such that:

1. t maps all yes instances of D1 to yes instances of D2 and all no instances of D1 to no instances of D2

2. t is computable by a polynomial time algorithm This definition immediately implies that if a problem D1 is polynomially reducible to some problemD2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time .

A decision problem D is said to be **NP-complete** if:

1. it belongs to class NP

2. every problem in NP is polynomially reducible to D

A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following. Nondeterministic (“guessing”) stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I.

Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial. Most decision problems are in NP. First of all, this class includes all the problems in P: $P \subseteq NP$.

5.2 COPING WITH THE LIMITATIONS OF ALGORITHM POWER

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.

5.3 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 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.

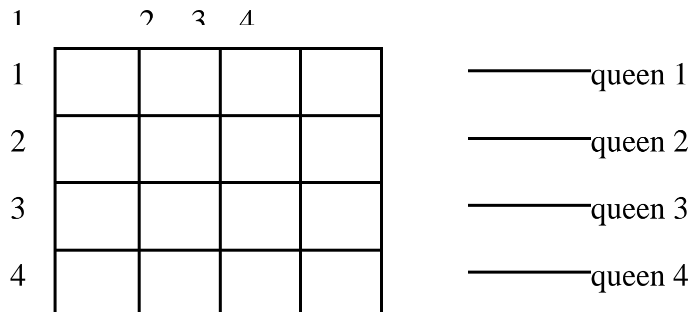
A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise it is called non

promising. Leaves represent either nonpromising dead ends or complete solutions found by the algorithms.

5.3.1 N-QUEENS PROBLEM

The problem is to place n Queens on an nXn 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



5.3.2 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.

5.3.3 SUBSET-SUM PROBLEM

Find a subset of a given set s={s1,………sn} 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:

$s1 \leq s2 \leq s3 \ldots\ldots\ldots sn$

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:

$s'+s_{i+1}>d$ (the sum s′ is too large)

$s'+\sum_{j=i+1}^{n} s_j<d$ (the sum s′ is too small)

It is generally used as 'n' tuples to generate the tree as x1, x2,x3………..xn

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€si+1 consistent with x[1…i] and the constraints do x[i+1] 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.

5.4 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 feasiblesolution 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.

5.4.1 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 a	9	2	7	8
Person b	6	4	3	7
Person c	5	8	1	8
Person d	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)

5.4.2 KNAPSACK PROBLEM

Given a items of known weights Wi and values Vi, i=1,2,........n and a knapsack of capacity W, find the most valuable subset of the items that items that fit in the knapsack.

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. This problem was introduced in Section 3.4: given n items of known weights wi and values vi , i = 1, 2, . . . , n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily: $v1/w1 \geq v2/w2 \geq . . . \geq vn/wn$.

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (see Figure 12.8 for an example). Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection. A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack W − w and the best per unit payoff among the remaining items, which is vi+1/wi+1: ub = v + (W − w)(vi+1/wi+1).

As a specific example, let us apply the branch-and-bound algorithm to the instance of the knapsack problem (We reorder the items in descending order of their value-to-weight ratios, though.)

item	weight	value	v/w
1	4	$40	10
2	7	$42	6
3	15	$20	5
4	3	$12	4

5.4.3 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.

5.5 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.

5.5.1 APPROXIMATION ALGORITHMS FOR THE TRAVELING SALESMAN PROBLEM

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.

Nearest-neighbor algorithm

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 = ∞ (unbounded above) – make the length of AD arbitrarily large in the above example

Multifragment-heuristic algorithm

Step 1 Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.
Step 2 Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.
Step 3 Return the set of tour edges.

There is, however, a very important subset of instances, called **Euclidean**, for which we can make a nontrivial assertion about the accuracy of both the nearestneighbor and multifragment-heuristic algorithms. These are the instances in which intercity distances satisfy the following natural conditions:

- triangle inequality $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities i, j, and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j)
- symmetry $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from ito j is the same as the distance from j to i)

Minimum-Spanning-Tree–Based Algorithms There are approximation algorithms for the traveling salesman problem that exploit a connection between Hamiltonian circuits and spanning trees of the same graph. Since removing an edge from a Hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straightforward fashion.

Twice-around-the-tree algorithm

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
Stage 2: Starting at an arbitrary vertex, create a path that 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 = ∞ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

5.5.2 APPROXIMATION ALGORITHMS FOR THE KNAPSACK PROBLEM

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: $v1/w1 \geq \ldots \geq vn/wn$

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, w1=1, v1=2, w2=m, v2=m)
yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values: $v1/w1 \geq \ldots \geq vn/wn$

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 knapsack, add the remaining items in that fit the decreasing.

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(knk+ 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.

Greedy algorithm for the discrete knapsack problem

Step 1 Compute the value-to-weight ratios ri= vi/wi, i = 1, . . . , n, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

Greedy algorithm for the continuous knapsack problem

Step 1 Compute the value-to-weight ratios vi/wi, i = 1, . . . , n, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list: if the current item on the list fits into the knapsack in its entirety, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.