DESIGN AND ANALYSIS OF ALGORITHMS

UNIT I: INTRODUCTION

Algorithm - Fundamentals of Algorithmic Problem Solving - Time Complexity - Space Complexity - Growth of Functions - Asymptotic notation: Need, Types – Big Oh, Little Oh, Omega, Theta – Properties - Complexity Analysis - Performance Measurement - Instance Size - Test Data - Experimental Setup.

Unit II: MATHEMATICAL FOUNDATIONS

Solving Recurrence equations - Substitution Method - Iterative Method - Recursion Tree Method - Master Method - Worst Case - Average Case - Best Case Analysis - Sorting in Linear Time - Lower bounds for sorting:- Counting Sort - Radix Sort – Bucket Sort.

Unit III: BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force Algorithm - Travelling Salesman Problem - Knapsack problem - Assignment Problem – Closest Pair and Convex Hull Problems - Divide and Conquer Approach:- Binary Search - Quick Sort - Merge Sort - Strassen's Matrix Multiplication.

Unit IV: GREEDY APPROACH ANDDYNAMIC PROGRAMMING

Greedy Approach - Optimal Merge Patterns - Huffman Code - Job Sequencing with Deadline - Tree Vertex Splitting - Dynamic Programming:- Dice Throw - Optimal Binary Search algorithms.

Unit V: BACKTRACKING AND BRANCH AND BOUND

Backtracking:- 8 Queens – Hamiltonian Circuit Problem – Branch and Bound:– Assignment Problem – Knapsack Problem – Travelling Salesman Problem – NP Complete Problems – Clique Problem – Vertex Cover Problem

References:

1. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Third Edition, PHI Learning Private Limited, 2012.
2. Alfred V.Aho, John E.Goldberg, "Genetic Algorithm In Search Optimization and Machine Learning", Pearson Education India, 2013.
3. Anany Levitin, "Introduction to the Design and Analysis of Algorithms", Third Edition, Pearson Education, 2012
4. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, "Fundamentals of Computer Algorithms", Second Edition, Universities Press, 2007

UNIT I:

1. INTRODUCTION

1.1 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.

1.1.1 Characteristics of an algorithm:

- **Input:** Zero / more quantities are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
- **Efficiency:** Every instruction must be very basic and runs in short time.

1.1.2 Example of Algorithm: Add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
            sum=num1+num2
Step 5: Display sum
Step 6: Stop
```

1.2 Fundamentals of Algorithmic Problem Solving

1. Understanding the problem.
2. Ascertaining the capabilities of a computational device.
3. Choosing between exact and approximate problem solving.
4. Deciding an appropriate Data Structure.
5. Algorithm design techniques.
6. Methods of specifying an algorithm.
7. Proving algorithms correctness.
8. Analyzing an algorithm.
9. Coding an algorithm.

1. **Understanding the problem**:
   - ✓ This is the first step in designing of algorithm.
   - ✓ Read the problem's description carefully to understand the problem statement completely.
   - ✓ Ask questions for clarifying the doubts about the problem.
   - ✓ Identify the problem types and use existing algorithm to find solution.
   - ✓ Input (instance) to the problem and range of the input get fixed.

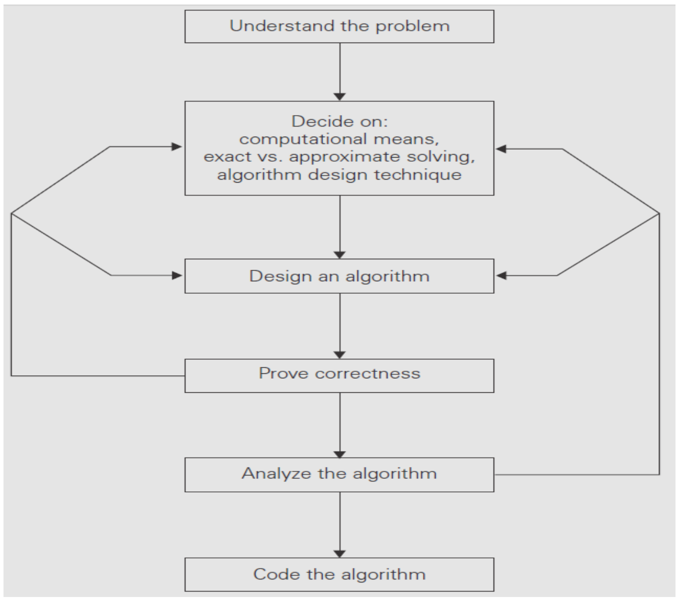


Figure 1.1 Fundamentals of Algorithmic Problem Solving

2. **Decision making:**
   a. **Ascertaining the capabilities of a computational device:**
      - ➢ In random-access machine (RAM), instructions are executed one after another (The central assumption is that one operation at a time). Algorithms designed to be executed on such machines are called sequential algorithms.
      - ➢ In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
      - ➢ Choice of computational devices like Processor and memory is mainly based on space and time efficiency.

b. Choosing between Exact and Approximate Problem Solving:

- ➢ An algorithm that can solve the problem exactly and produce correct result is called an exact algorithm.
- ➢ The algorithm that can solve the problem approximately and not able to produce exact solution is called an approximation algorithm.

  Example: extracting square roots, solving nonlinear equations, and evaluating definite integrals.

c. Algorithm Design Techniques:

- ▪ An **algorithm design technique** is a general approach to solving problems algorithmically. It is suitable for a variety of problems from different areas of computing.
  - **Programs = Algorithms + Data Structures**
- ▪ Algorithms and Data Structures are independent, but they are combined together to develop program. The selection of proper data structure is necessary before designing the algorithm.
- ▪ Some of the design techniques are Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

3. Methods of specifying an algorithm:

There are 3 methods:

1. **Natural language**:

- ➢ It is Easy but ambiguous. Such a specification creates difficulty while actually implementing it. So many programmers prefer to have specification of algorithm by means of Pseudocode.

2. **Pseudo code**:

- ➢ It is a Mixture of programming language and natural language.

  Pseudocode is usually more precise than natural language.

3. **Flow chart**:

- ➢ It is a pictorial representation of the algorithm. Here symbols are used. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps.

4. **Proving algorithms correctness:**

The algorithm yields a required result for every legitimate input in a finite amount of time. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. For an approximation algorithm, the error produced by the algorithm does not exceed a predefined limit.

5. Analyzing an algorithm:

Algorithm analysis is an important part and it is the determination of the amount of time and space resources required to execute it.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance.

Types of Algorithm Analysis:

1. **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm.
2. **Worst case:** Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm.
3. **Average case:** In the average case take all random inputs and calculate the computation time for all inputs.

6. Coding an algorithm:

The coding / implementation of an algorithm is done by a suitable programming language. It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

1.3 Time Complexity

- The time required by the algorithm to solve given problem is called **time complexity** of the algorithm. Time complexity is very useful measure in algorithm analysis.
- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- It is important to note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.
- Time complexity or Big O notation is typically written as $T(n)$, where n is a variable related to the size of the input.

There are different types of time complexities used, such as:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

1. **Constant time, or *O*(1),** is the time complexity of an algorithm that always uses the same number of operations, regardless of the number of elements being operated on.
2. **Linear time, or *O*(*n*),** indicates that the time it takes to run an algorithm grows in a linear fashion as *n* increases.
3. **Logarithmic time, or *O*(log *n*),** indicates that the time needed to run an algorithm grows as a logarithm of *n*.
4. **Quadratic time, or $O(n^2)$,** indicates that the time it takes to run an algorithm grows as the square of *n*.

Example: Addition of two numbers.

```
Algorithm ADD(A, B)
//Perform arithmetic addition of two numbers
//Input: Two variables A and B
//Output: variable C, which holds the addition of A and B
C = A + B
return C
```

The addition of two numbers requires one addition operation. The time complexity of this algorithm is constant, so T(n) = O(1) .

1.4 Space Complexity

The **space complexity** is the measurement of total space required by an algorithm to execute properly.

It also includes memory required by input variables. Basically, it's the sum of auxiliary space and the memory used by input variables.

Space complexity = Auxiliary space + Memory used by input variables

Auxiliary Space is the extra space or temporary space used by an algorithm.

An algorithm's space can be categorized into 2 parts:

1) Fixed Part - It is independent of the characteristics of input and output. It includes instruction (code) space, space for simple variables, fixed-size component variables and constants.

2) Variable Part - It depends on instance characteristics. It consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables, and the recursion stack space.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n*n, this will require $O(n^2)$ space.

There are many notations available to compute space complexity:

- **Big-O notation : OO:**

  This big-O notation describes the asymptotic upper bound, the worst case of space complexity. It's the measure of the maximum amount of space the algorithm takes to grow with respect to the size of input data.

- **Omega Notation: ΩΩ**

  The omega notation describes the asymptotic lower bound. It is the best-case scenario in terms of time and space complexity. Actually, it's the minimum amount of space our algorithm takes to grow with respect to the input data.

- **Theta Notation: θθ**

  The **theta notation** means that the space complexity of an algorithm is between its upper-bound and lower-bound space complexity.

Consider the following algorithm:

```
public int sum(int a, int b)
 {
   return a + b;
}
```

In this particular method, three variables are used and allocated in memory:

1. The first int argument, a
2. The second int argument, b
3. The returned sum result which is also an int like a and b

In Java, a single integer variable occupies 4 bytes of memory. In this example, there are three integer variables. Therefore, this algorithm always takes 12 bytes of memory to complete (3*4 bytes).

The space complexity is constant, so, it can be expressed in big-O notation as O (1).

1.5 Growth of Functions:

The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also which helps to compare the relative performance of alternative algorithms. The input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is **asymptotically** more efficient will be the best choice for all but very small inputs.

1.6 Asymptotic notation:

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken)

1.6.1 Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f\ (n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f\ (n)$ is said to be **asymptotically equivalent** to n^2 as $n \rightarrow \infty$", and here is written symbolically as $f\ (n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

1.6.2 Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

1.6.3 Asymptotic Notations:

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. Big-oh notation:

Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

f (n) ⩽ k.g (n)f(n)⩽k.g(n) for n>n0n>n0 in all case

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)

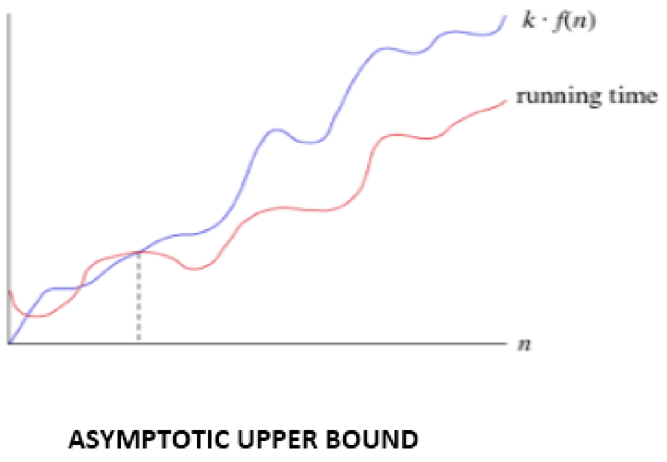


Figure 1.2 Big-oh notation

Example:

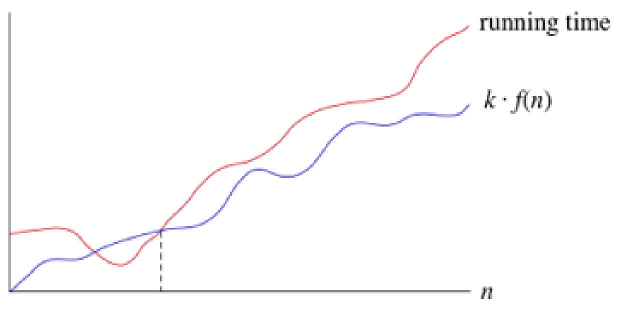
1. 1. 3n+2=O(n) as 3n+2≤4n for all n≥2
2. 2. 3n+3=O(n) as 3n+3≤4n for all n≥3

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega (Ω) Notation:

The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

F (n) ≥ k* g (n) for all n, n≥ n_0



ASYMPTOTIC LOWER BOUND

Figure 1.3 Omega (**Ω**) Notation

Example:

f (n) $=8n^2+2n-3 \geq 8n^2-3$
$=7n^2+(n^2-3) \geq 7n^2$ (g(n))

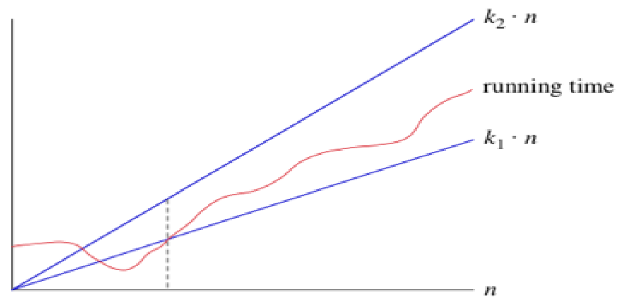
Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as Ω (g (n))

3. Theta (θ):

The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC TIGHT BOUND

Figure 1.4 Theta (θ) notation

Example:

$3n+2= \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n

$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

1.7 Complexity Analysis

Typical Complexities of an Algorithm

- **Constant Complexity:**

  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.

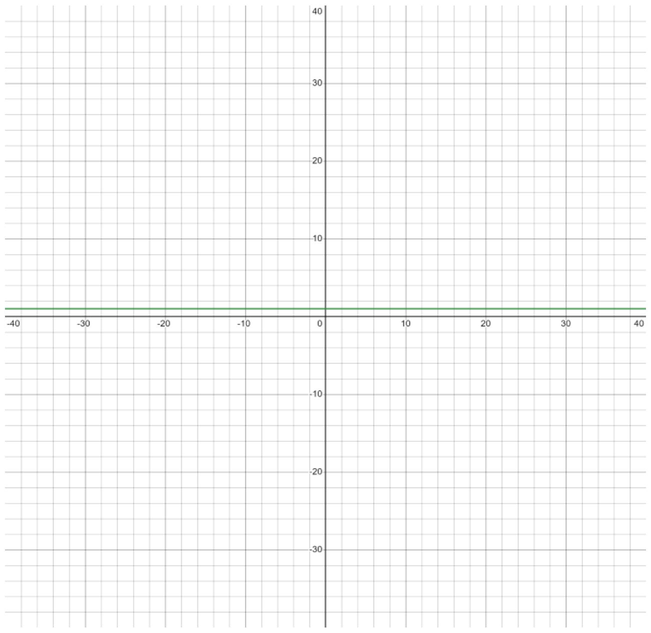


Figure 1.5 Constant Complexity

- **Logarithmic Complexity:**

  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log (N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.

  For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.

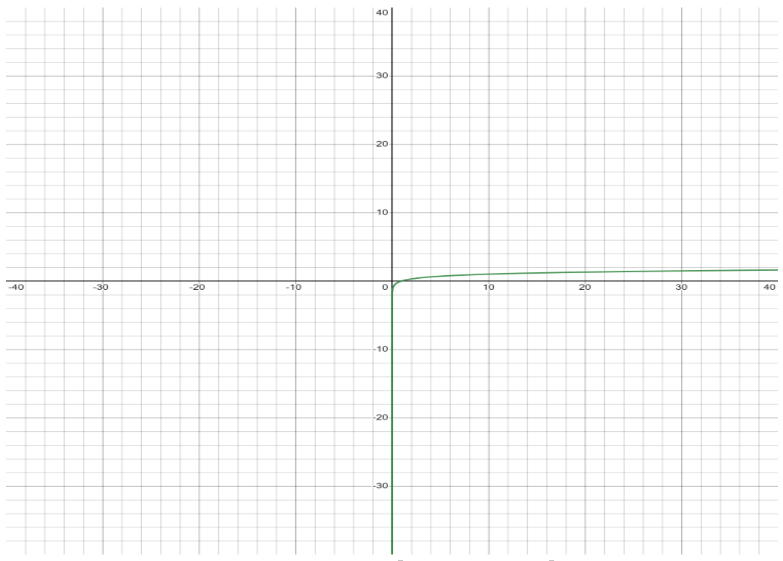


Figure 1.6 Logarithmic Complexity

- **Linear Complexity:**

  It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.

  For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.

  It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem.

  For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

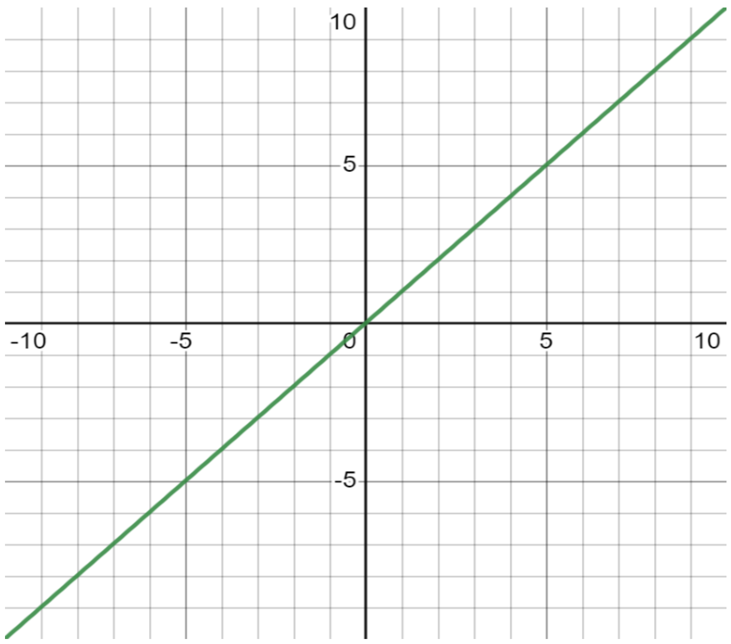


Figure 1.7 Linear Complexity

- **Quadratic Complexity:**

  It imposes a complexity of **O(n^2)**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.

  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity.

  For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.

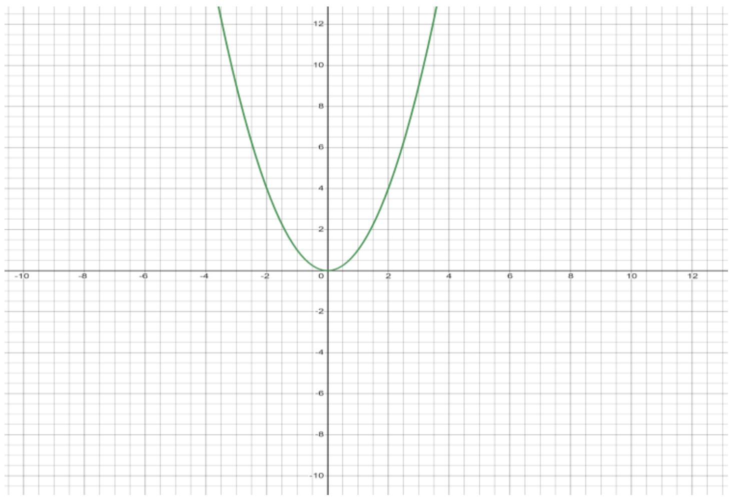


Figure 1.8 Quadratic Complexity

- **Cubic Complexity:**

  It imposes a complexity of **O(n^3)**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.

For example, if there exist 100 elements, it is going to execute 1,000,000 steps.

- **Exponential Complexity:**

  It imposes a complexity of **$O(2^n)$, $O(N!)$, $O(n^k)$, ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.

  For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

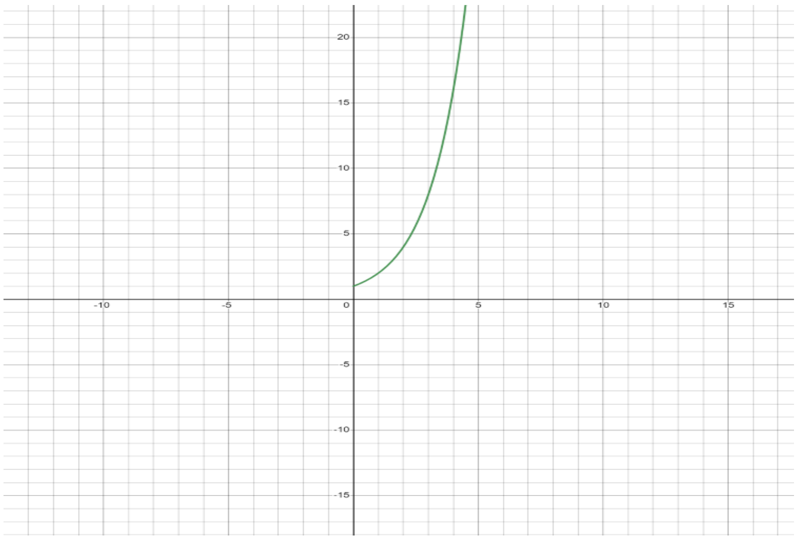


Figure 1.9 Exponential Complexity

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them. Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem.

1.8 Performance Measurement

In computer science, there are multiple algorithms to solve a problem. Suppose there is more than one algorithm is available to solve a problem, it is necessary to select the best one. Performance analysis helps to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem. We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements. Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

The following metrics are used to evaluate the performance of an algorithm:

- The amount of space necessary to perform the algorithm’s task (Space Complexity). It consists of both program and data space.
- The time necessary to accomplish the algorithm’s task (Time Complexity).

Notation of Performance Measurement

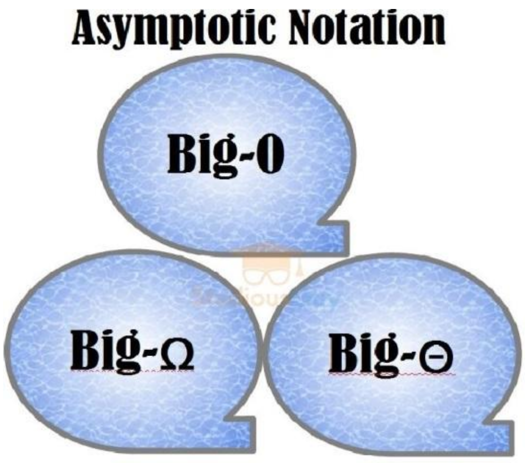


Figure 1.10 The Notation for Performance Measurement of an Algorithm

The Notation is termed Asymptotic Notation and is a mathematical representation of the algorithm's complexity. The following are three asymptotic notations for indicating time-complexity each of which is based on three separate situations, namely, the best case, worst case, and average case:

a. **Big – O (Big-Oh)**

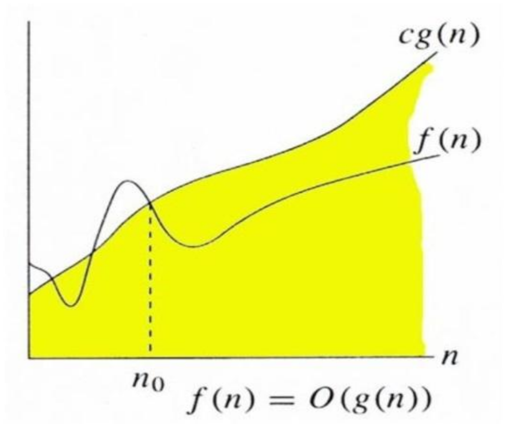


Figure 1.11 Big - O

The top-bound of an algorithm's execution time is represented by the Big-O notation. It's the total amount of time an algorithm takes for all input values. It indicates an algorithm's worst-case time complexity.

b. **Big – Ω (Omega)**

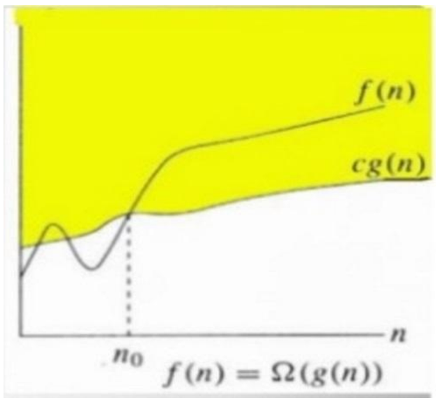


Figure 1.12 Big - Ω

The omega notation represents the lowest bound of an algorithm’s execution time. It specifies the shortest time an algorithm requires for all input values. It is the best-case scenario for the time complexity of an algorithm.

c. **Big – Θ (Theta)**

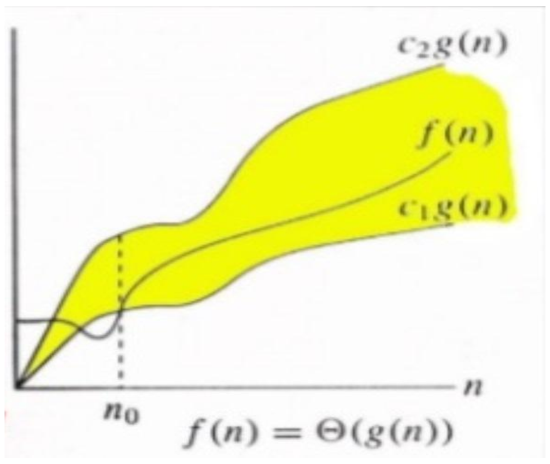


Figure 1.13 Big - Θ

The function is enclosed in the theta notation from above and below. It reflects the average case of an algorithm’s time complexity and defines the upper and lower boundaries of its execution time.

1.9 Instance Size

- The instance size refers to the input size or the scale of the problem being considered by the algorithm.
- For example, in the context of sorting algorithms, the instance size might be the number of elements in the array to be sorted. In graph algorithms, it could be the number of vertices and edges.
- Varying the instance size allows you to observe how the algorithm's performance scales with the size of the problem.
-

1.10. Test Data

- Test data is the specific input values or datasets used to evaluate the algorithm's behavior and performance.
- It's essential to use a diverse set of test cases that cover a range of scenarios and input configurations. This helps ensure that the algorithm performs well in various situations.
- Test data may include best-case, average-case, and worst-case scenarios to provide a comprehensive understanding of the algorithm's efficiency under different conditions.

1.11. Experimental Setup

- The experimental setup includes all the details and conditions under which the algorithm is tested and analysed.
- Parameters such as the hardware (e.g., CPU, memory), software environment, programming language, and compiler settings should be specified.
- The experimental setup should be well-documented and reproducible to allow others to verify and replicate the results.
- The choice of programming language and compiler can impact the algorithm's performance, so it's crucial to provide this information.

SAMPLE QUESTIONS:

1. Define an algorithm and explain its key characteristics.
2. Explain the difference between best-case, worst-case, and average-case time complexities. Provide an example for each.
3. Analyze the space complexity of a recursive algorithm to find the factorial of a number.
4. Compare and contrast time complexity and space complexity. How can an algorithm be optimized in terms of space complexity?
5. Define Big O notation and explain its significance in analyzing algorithmic complexity.
6. Given two functions f(n) = 2n^2 + 3n and g(n) = n^2 + 5, determine whether f(n) is O(g(n)).
7. Discuss the importance of complexity analysis in the design and evaluation of algorithms.
8. Compare and contrast time complexity and computational complexity. Provide examples to illustrate the concepts.

9. Explain the concept of empirical analysis in the context of algorithm performance measurement.
10. How does the instance size affect the performance of an algorithm? Provide examples to illustrate your answer.
11. Describe the role of test data in evaluating the efficiency of an algorithm. How can different types of data impact performance?

UNIT II: Mathematical Foundations

2.1. Solving Recurrence equations

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

Example: The worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

$T(n)= \Theta(1)$; if $n=1$

$2T(n/2) + \Theta(n)$;if $n>1$

whose solution can be found as $T(n)=\Theta(n\log n)$

There are various techniques to solve recurrences.

2.2 Substitution Method

The substitution method comprises of 3 steps

i. Guess the form of the solution

ii. Verify by induction

iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence it is called “substitution method”. This method is powerful, but we must be able to guess the form of the answer in order to apply it.

Example: recurrence equation: $T(n)=4T(n/2)+n$

step 1: guess the form of solution

$T(n)=4T(n/2)$

- ➢ $F(n)=4f(n/2)$
- ➢ $F(2n)=4f(n)$
- ➢ $F(n)=n^2$

So, T(n) is order of n^2

Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$

$T(n)=4T(n/2)+n$

$<=4c(n/2)^3 +n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 –n)$ is always positive

So what we assumed was true.

- $T(n)=O(n^3)$

Step 3: solve for constants

$Cn^3/2-n>=0$

- $n>=1$
- $c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume $(k)<=ck^2$

So, we should prove that $T(n)<=cn^2$

$T(n)=4T(n/2)+n$

- $4c(n/2)^2+n$
- cn^2+n

So, T(n) will never be less than cn2. But if we will take the assumption of T(k)=c1 k2-c2k, then we can find that T(n) = O(n2)

2.3 Iterative Method

Example:

$T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^3) +3n$

After k iterations, T(n)=2kT(n/2k)+kn--------------(1)

Sub problem size is 1 after n/2k=1 => k=logn

So,after logn iterations ,the sub-problem size will be 1.

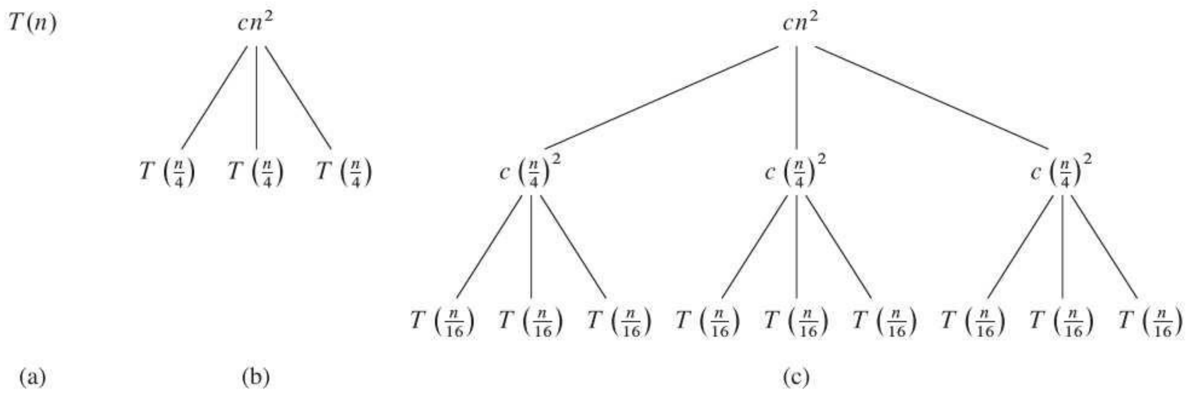
So,when k=logn is put in equation 1

T(n)=nT(1)+nlogn

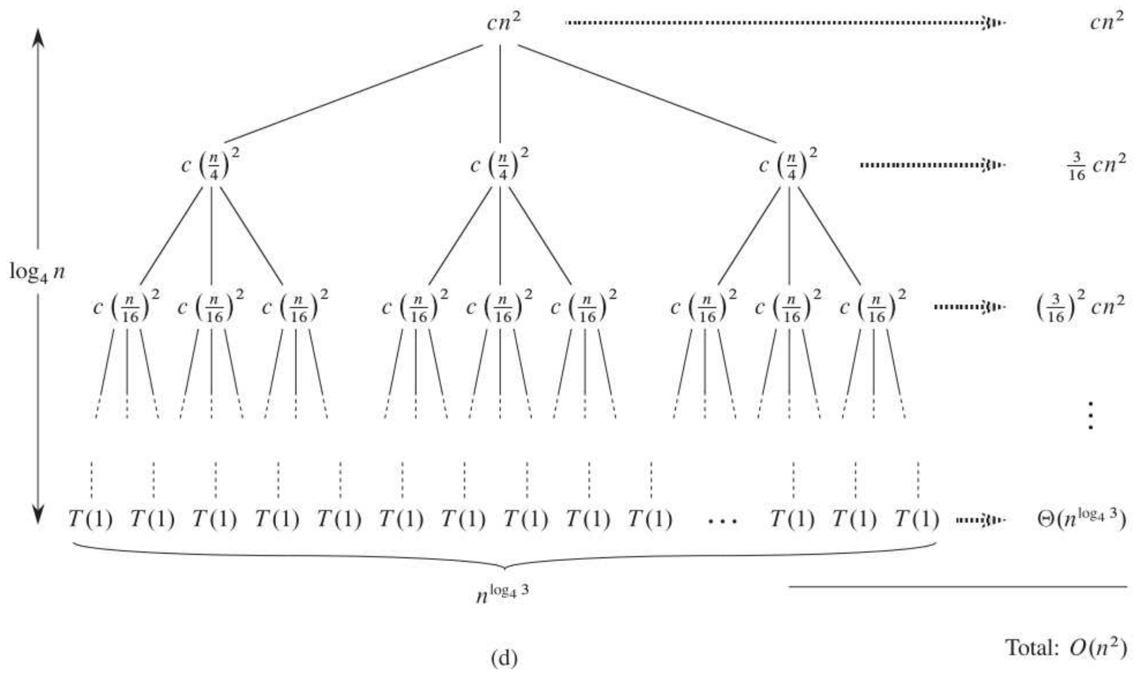
- nc+nlogn (say c=T(1))
- O(nlogn)

2.4 Recursion Tree Method

- In a recursion tree, each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost, and then we sum all the per level cost to determine the total cost of all levels of recursion.
- Constructing a recursion tree for the recurrence T(n)=3T(n/4)+cn^2



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)



Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part(d) has height log4n (it has log4n + 1 levels).

Sub problem size at depth i =$n/4^i$

Sub problem size is 1 when $n/4^i=1 \Rightarrow i=\log_4 n$

So, no. of levels =1+ $\log_4 n$

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth i=3^i

Cost of each node at depth i=$c\,(n/4^i)^2$

Cost of each level at depth i=$3^i\,c\,(n/4^i)^2 = (3/16)^i cn^2$

T(n)= $_{i=0}\Sigma^{\log_4 n}\, cn^2(3/16)^i$

T(n)= $_{i=0}\Sigma^{\log_4 n - 1}\, cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level =$3^i T(1)$

- ➢ $c3^{\log_4 n}$ (at last level i=log4n)
- ➢ $cn^{\log_4 3}$

T(n)= $_{i=0}\Sigma^{\log_4 n - 1}\, cn^2(3/16)^i + c\, n^{\log_4 3}$

$<= cn^{2\,*}(16/13)+ c\, n^{\log_4 3} \Rightarrow T(n)=O(n^2)$

2.5 Master Method

The master method solves recurrences of the form

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

where a>=1 and b>1 are constants and f(n) is a asymptotically positive function .

The Master Theorem provides a solution in terms of big-O notation based on the form of f(n). The solution depends on how f(n) compares to certain functions.

To use the master method, we have to remember **3 cases**:

1. If $f(n)=O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon > 0$,then T(n)=Θ(nlogba)
2. If $f(n)=\Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a}\log n)$
3. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$,and if a*f(n/b)<=c*f(n) for some constant c<1 and all sufficiently large n,then T(n)=Θ(f(n))

Example:

T(n)=2T(n/2)+nlogn

a=2, b=2, and f(n)=nlogn

using case 2

$f(n)=\Theta(n^{\log_2 2}\log^k n)$

$=>\Theta(n^1 \log^k n)=n\log n$ $=>K=1$

$T(n)=\Theta(n^{\log_2 2} \log^1 n)$

$=>\Theta(n\log^2 n)$

2.6 Worst Case, Average Case and Best Case Analysis:

The worst, average, and best case of an algorithm depends on the size of the user input value.

2.6.1 Worst Case Analysis:

In the worst-case analysis, calculate the upper limit of the execution time of an algorithm. It is necessary to know the case which causes the execution of the maximum number of operations.

2.6.2 Average Case Analysis:

In the average case analysis, take all possible inputs and calculate the computation time for all inputs. Add up all the calculated values and divide the sum by the total number of entries.

2.6.3 Best Case Analysis:

In the best case analysis, calculate the lower bound of the execution time of an algorithm. It is necessary to know the case which causes the execution of the minimum number of operations.

2.6.4 Example: Worst Case, Average Case and Best Case Analysis:

Merge sort is a sorting algorithm that is trivial to apply and has a time complexity of $O(n*\log n)$ for all conditions (best case, worst case and average case). This algorithm is based on the divide and conquers strategy.

The sorting algorithm continuously splits a list into multiple sublists until each sublist has only one item left. It then merges those sublists into a sorted list.

What is Merge Sort?

Merge Sort, considered to be the most efficient sorting algorithm, works on the principle '**Divide and Conquer**'.

The algorithm can be divided into three parts:

1. **Divide** - It is the initial stage where the **midpoint** of the array is found using mid=start+(end−start)/2

2. **Conquer** - In this step, the array is divided into subarrays, using the midpoint calculated. The process repeats itself recursively until all elements become single array elements.
3. **Combine** - In this step, the **subarrays** formed are combined in **sorted order**.

2.6.4.1 Complexity Analysis of Merge Sort

Merge sort repeatedly divides the array into **two equally sized parts**.

Thus merge sort time complexity depends on the number of division stages.

The number of division stages is log 2 n.

On each merge stage, n elements are merged.

- Step 1 - n×1
- Step 2 - n/2×2
- Step 3 - n/4×4

Merge Sort time complexity is calculated using time per division stage. Since the merge process has linear time complexity, for n elements there will be n∗log 2 n division and merge stages.

Hence, regardless of the arrangement, the time complexity of Merge Sort is O(nlogn)

2.6.4.2 Analysis of Merge Sort Time Complexity

a. **Best Case Time Complexity of Merge Sort**

The best case scenario occurs when the elements are already sorted in **ascending order**.

If two sorted arrays of size n need to be merged, the minimum number of comparisons will be n. This happens when all elements of the first array are less than the elements of the second array. The best case time complexity of merge sort is O(n∗logn).

b. **Average Case Time Complexity of Merge Sort**

The average case scenario occurs when the elements are **jumbled** (neither in ascending nor descending order). This depends on the number of comparisons. The average case time complexity of merge sort is O(n∗logn).

c. **Worst Case Time Complexity of Merge Sort**

The worst-case scenario occurs when the given array is sorted in **descending order** leading to the maximum number of comparisons. In this case, for two sorted arrays of size n, the minimum number of comparisons will be 2n. The worst-case time complexity of merge sort is O(n∗logn).

2.6.4.3 Analysis of Merge Sort Space Complexity

In merge sort, all elements are copied into an **auxiliary array** of size N, where N is the **number of elements present in the unsorted array**. Hence, the space complexity for **Merge Sort** is O(N).

Conclusion

- Merge sort is a reliable sorting algorithm that uses a '**divide and conquers**' paradigm.
- Initially, it divides the problem into sub-problems and solves them individually.
- Then, it combines the results of sub-problems to get the solution to the original problem.
- Merge sort is a **recursive sorting algorithm**.
- The recurrence relation of merge sort is T(n)=2T(n/2)+θ(n).
- The time complexity of merge sort is
  - Best Case: O(n∗logn)
  - Average Case: O(n∗logn)
  - Worst Case: O(n∗logn)O
- The space complexity of **merge sort** is O(n).

2.7 Sorting In Linear Time

There are several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heap sort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

In these algorithms the sorted order is determined based only on comparisons between the input elements. These types of sorting algorithms are called as ***comparison sorts***. Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort a sequence of n elements. Thus, merge sort and heap sort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

2.8 Lower bounds for sorting

Comparison sort, use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, . . . ,a_n)$ That is, given two elements a_i and a_j, perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, *or* $a_i > a_j$ to determine their relative order. This sorting not inspects the values of the elements or gain order information about them in any other way.

Let as assume without loss of generality that all of the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so assume that no comparisons of this form are made. Also note that the comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. Therefore assume that all comparisons have the form $a_i \leq a_j$.

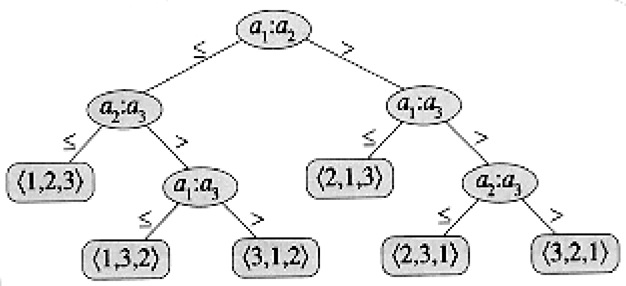


Figure 2.1

The decision tree for insertion sort operating on three elements. There are 3! = 6 possible permutations of the input elements, so the decision tree must have at least 6 leaves.

2.8.1The decision-tree model

Comparison sorts can be viewed abstractly in terms of ***decision trees***. A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. The above Figure shows the decision tree corresponding to the insertion sort algorithm from operating on an input sequence of three elements.

In a decision tree, each internal node is annotated by $a_i : a_j$ for some i and j in the range $1 <= i, j <= n$, where n is the number of elements in the input sequence. Each leaf is annotated by a permutation $\langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle$. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i <= a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i <= a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\Pi(1)} \leq a_{\Pi(2)} \leq \ldots \leq a_{\Pi(n)}$. Each of the n! permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

2.8.2 A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons the sorting algorithm performs. Consequently, the worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 1

Any decision tree that sorts n elements has height Ω(n lg n).

Proof: Consider a decision tree of height *h* that sorts *n* elements. Since there are *n*! permutations of *n* elements, each permutation representing a distinct sorted order, the tree must have at least *n*! leaves. Since a binary tree of height *h* has no more than 2^h leaves, we have

$$n! \leq 2^h ,$$

which, by taking logarithms, implies

$$h \geq \log(n!) ,$$

Since the log function is monotonically increasing. From Stirling's approximation, we have

$$n! > \left(\frac{n}{e}\right)^n ,$$

where *e* = 2.71828 . . . is the base of natural logarithms; thus

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

Corollary: Heap sort and merge sort are asymptotically optimal comparison sorts.

Proof: The $O(n \log n)$ upper bounds on the running times for heap sort and merge sort match the $\Omega(n \log n)$ worst-case lower bound from Theorem 1.

2.9 Counting sort

Counting Sort was discovered by Harold Seward in 1954. Counting sort is a linear sorting algorithm with asymptotic complexity O(n+k). The Counting Sort method is a fast and reliable sorting algorithm. Counting sort is not a comparison-based algorithm. It avoids comparisons and takes advantage of the array's O(1) time insertions and deletions.

Counting sort assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When k = O(n), the sort runs in O(n) time.
Counting sort is a sorting technique that is based on the keys between specific ranges. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values.

Working of counting sort Algorithm

Now, let's see the working of the counting sort Algorithm.

To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

Let the elements of array are -

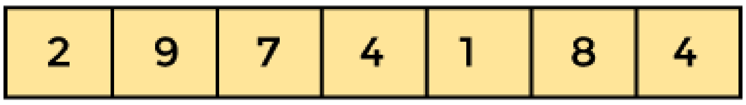


Figure 2.2

1. Find the maximum element from the given array. Let **max** be the maximum element.

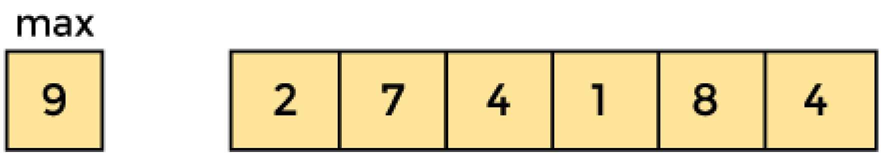


Figure 2.3

2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

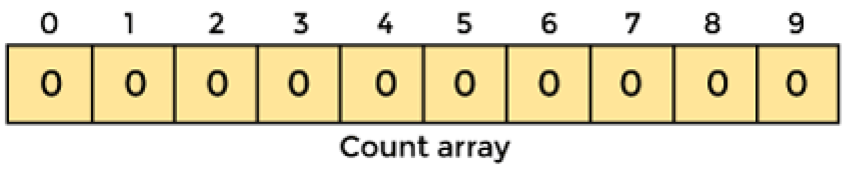


Figure 2.4

3. Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.

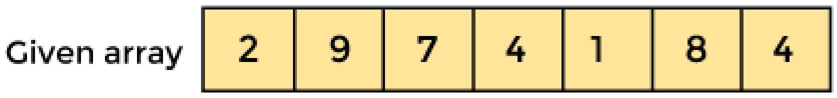


Figure 2.5

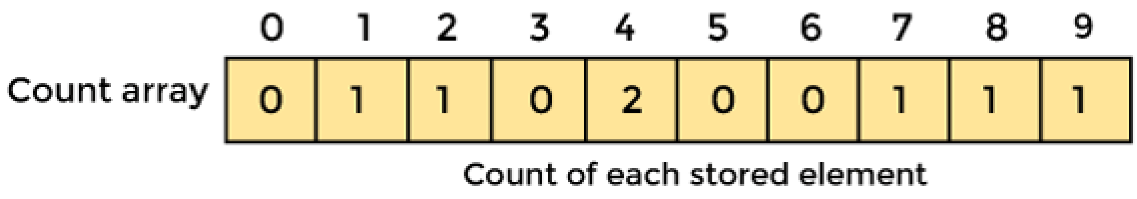


Figure 2.6

Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.

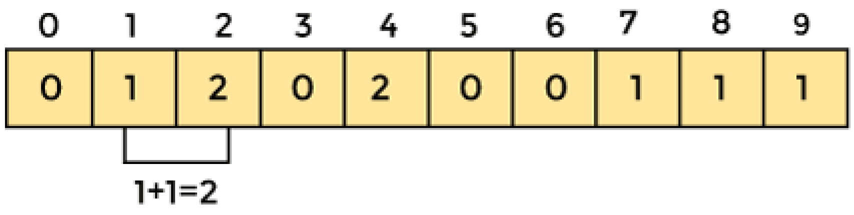


Figure 2.7

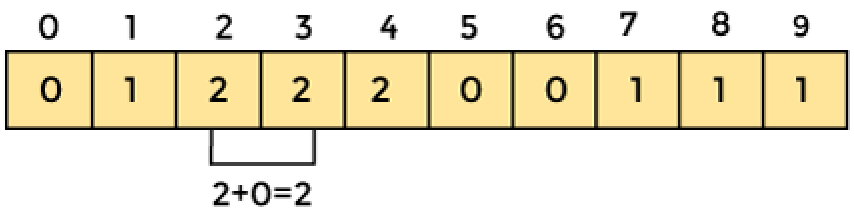


Figure 2.8

Similarly, the cumulative count of the count array is –

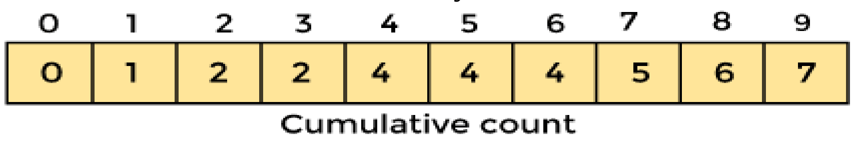


Figure 2.9

4. Now, find the index of each element of the original array

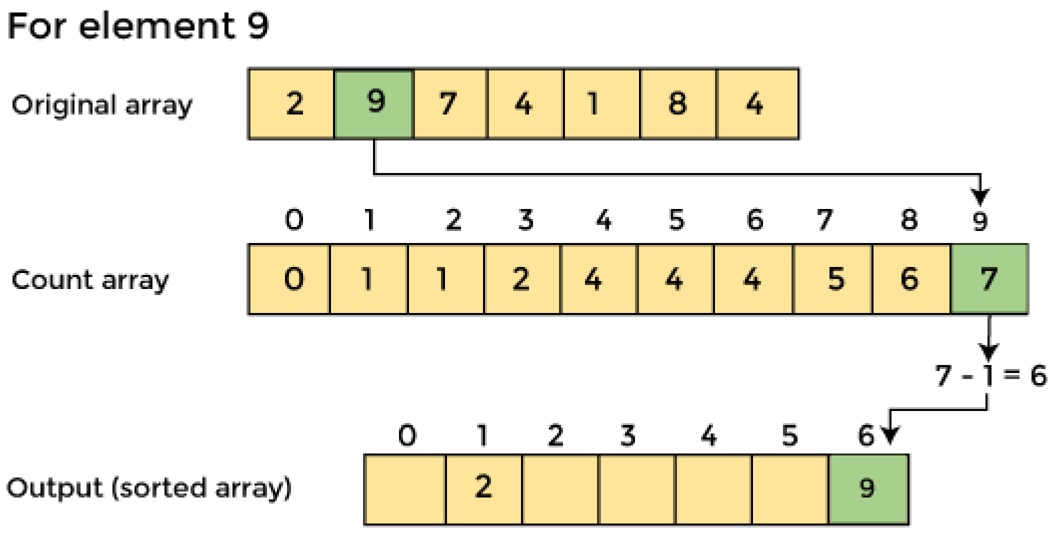


Figure 2.10

After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

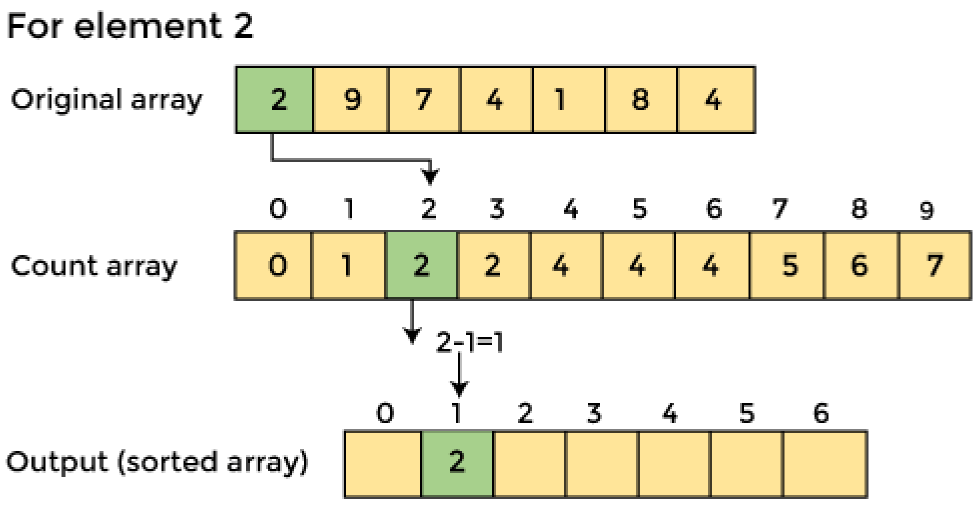


Figure 2.11

Similarly, after sorting, the array elements are –

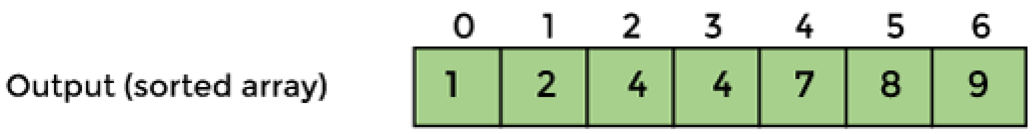


Figure 2.12

Now, the array is completely sorted.

2.9.1 Complexity Analysis of Counting Sort:

- **Time Complexity**: O(N+M), where **N** and **M** are the size of **inputArray[]** and **countArray[]** respectively.
    - Worst-case: O(N+M).
    - Average-case: O(N+M).
    - Best-case: O(N+M).
- **Auxiliary Space:** O(N+M), where **N** and **M** are the space taken by **outputArray[]** and **countArray[]** respectively.

2.9.2 Advantages of Counting Sort

- Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of input.
- Counting sort is easy to code
- Counting sort is a **stable algorithm**.

2.9.3 Disadvantages of Counting Sort

- Counting sort doesn’t work on decimal values.
- Counting sort is inefficient if the range of values to be sorted is very large.
- Counting sort is not an **In-place sorting** algorithm, It uses extra space for sorting the array elements.

Algorithm Counting Sort:

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

2.10 Radix sort

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys. Radix sort algorithm is a non-comparative sorting algorithm in computer science. It avoids comparison by creating and categorizing elements based on their radix.

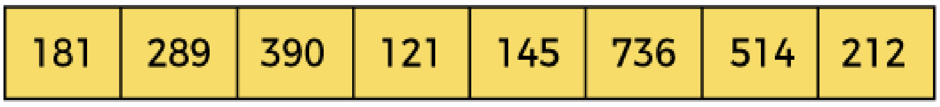
Algorithm

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. for i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

In a typical computer, which is a sequential random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

2.10.1 Working of Radix sort Algorithm

- First, to find the largest element (suppose **max**) from the given array. Suppose **'x'** be the number of digits in **max**. The **'x'** is calculated because it is needed to go through the significant places of all elements.
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.



181	289	390	121	145	736	514	212

Figure 2.13

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

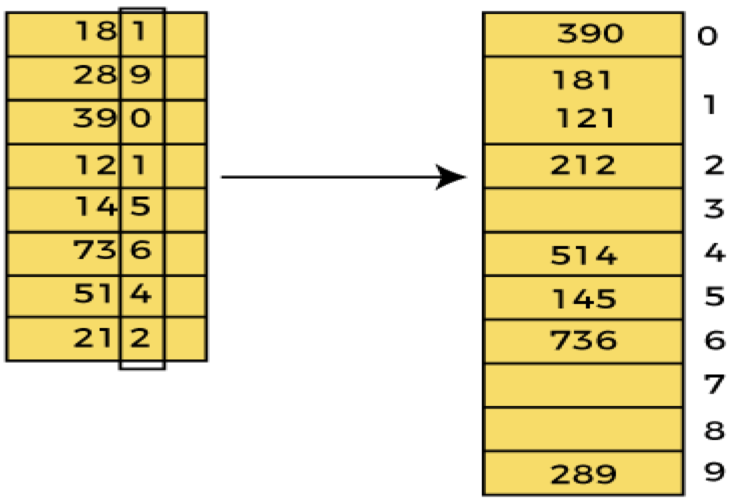
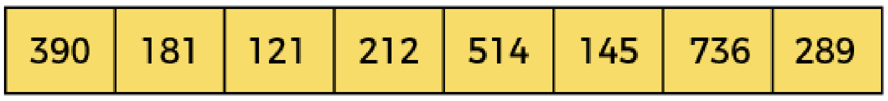


Figure 2.14

After the first pass, the array elements are –



390	181	121	212	514	145	736	289

Figure 2.15

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

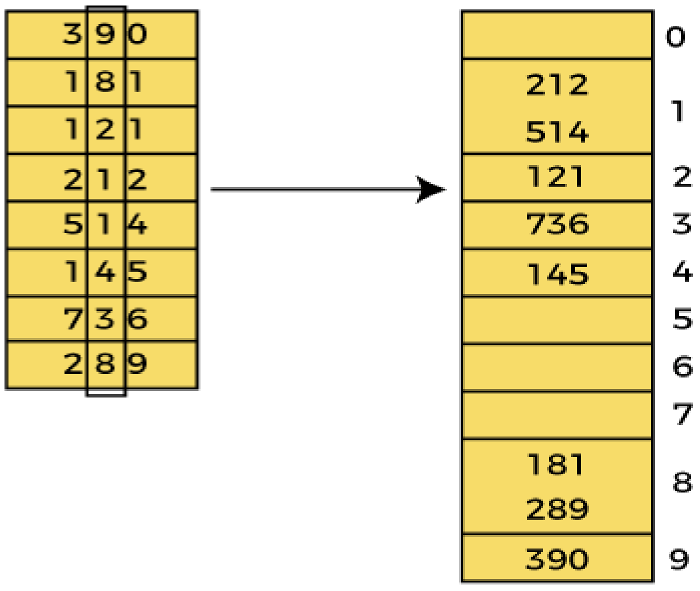


Figure 2.16

After the second pass, the array elements are -

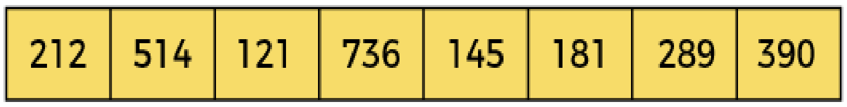


Figure 2.17

Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).

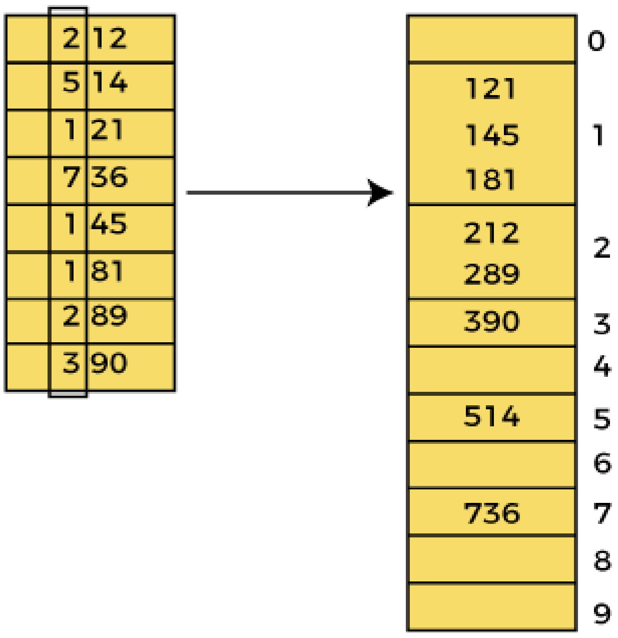


Figure 2.18

After the third pass, the array elements are –

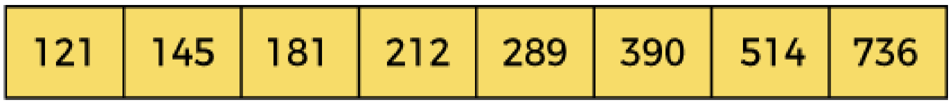


Figure 2.19

Now, the array is sorted in ascending order.

2.10.2 Radix sort complexity

a) **Time Complexity**

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk)**.

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk)**.

  Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

b) Space Complexity

The space complexity of Radix sort is O(n + k). Radix sort employs Counting sort, which uses auxiliary arrays of sizes n and k, where n is the number of elements in the input array and k is the largest element among the dth place elements (ones, tens, hundreds, and so on) of the input array. Hence, the Radix sort has a space complexity of (n+k).

2.11 Bucket Sort

It is a sorting technique that involves dividing elements into various groups, or buckets. These buckets are formed by uniformly distributing the elements. Once the elements are divided into buckets, they can be sorted using any other sorting algorithm. Finally, the sorted elements are gathered together in an ordered fashion.

The basic procedure of performing the bucket sort is given as follows –

- First, partition the range into a fixed number of buckets.
- Then, toss every element into its appropriate bucket.
- After that, sort each bucket individually by applying a sorting algorithm.
- And at last, concatenate all the sorted buckets.

Create **n** empty buckets (Or lists) and do the following for every array element arr[i].

- Insert arr[i] into bucket[n*array[i]]
- Sort individual buckets using insertion sort.
- Concatenate all sorted buckets.

Algorithm

```
Bucket Sort(A[])

1. Let B[0....n-1] be a new array

2. n=A.length[A]

3. for i=0 to n-1
```

```
4. make B[i] an empty list
5. for i=1 to n
6. do insert A[i] into list B[n a[i]]
7. for i=0 to n-1
8. do sort list B[i] with insertion-sort
9. Concatenate lists B[0], B[1],........, B[n-1] together in order
End
```

2.11.1 Working steps for Bucket Sort Algorithm

Step 1 − Divide the interval in ‘n’ equal parts, each part being referred to as a bucket. Say if n is 10, then there are 10 buckets; otherwise more.

Step 2 − Take the input elements from the input array A and add them to these output buckets B based on the computation formula, $B[i] = \lfloor n.A[i] \rfloor$

Step 3 − If there are any elements being added to the already occupied buckets, created a linked list through the corresponding bucket.

Step 4 − Then we apply insertion sort to sort all the elements in each bucket.

Step 5 − These buckets are concatenated together which in turn is obtained as the output.

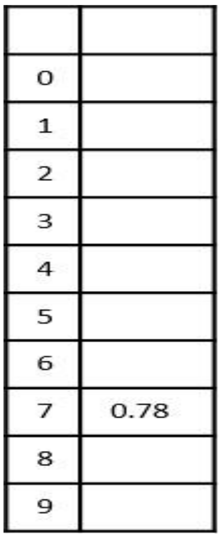
Example

Consider, an input list of elements, 0.78, 0.17, 0.93, 0.39, 0.26, 0.72, 0.21, 0.12, 0.33, 0.28, to sort these elements using bucket sort −

Solution

Step 1

Linearly insert all the elements from the index ‘0’ of the input array. That is, we insert 0.78 first followed by other elements sequentially. The position to insert the element is obtained using the formula − $B[i]= \lfloor n.A[i] \rfloor$, i.e, $\lfloor 10 \times 0.78 \rfloor=7$

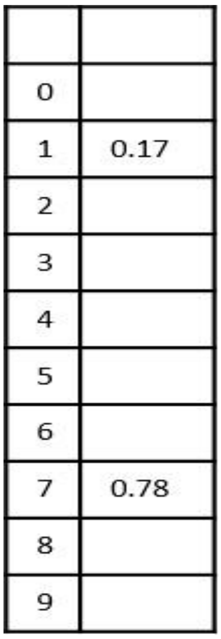


0	
1	
2	
3	
4	
5	
6	
7	0.78
8	
9	

Figure 2.20

Step 2

Now, we insert 0.17 at index $\lfloor 10 \times 0.17 \rfloor = 1$



0	
1	0.17
2	
3	
4	
5	
6	
7	0.78
8	
9	

Figure 2.21

Step 3

Inserting the next element, 0.93 into the output buckets at $\lfloor 10 \times 0.93 \rfloor = 9$

0	
1	0.17
2	
3	
4	
5	
6	
7	0.78
8	
9	0.93

Figure 2.22

Step 4

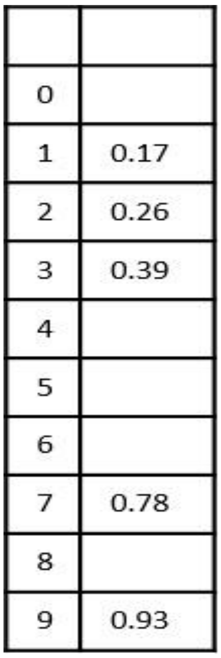
Insert 0.39 at index 3 using the formula $\lfloor 10 \times 0.39 \rfloor = 3$

0	
1	0.17
2	
3	0.39
4	
5	
6	
7	0.78
8	
9	0.93

Figure 2.23

Step 5

Inserting the next element in the input array, 0.26, at position $\lfloor 10 \times 0.26 \rfloor = 2$



0	
1	0.17
2	0.26
3	0.39
4	
5	
6	
7	0.78
8	
9	0.93

Figure 2.24

Step 6

Here is where it gets tricky. Now, the next element in the input list is 0.72 which needs to be inserted at index '7' using the formula $\lfloor 10 \times 0.72 \rfloor = 7$. But there's already a number in the 7th bucket. So, a link is created from the 7th index to store the new number like a linked list, as shown below –

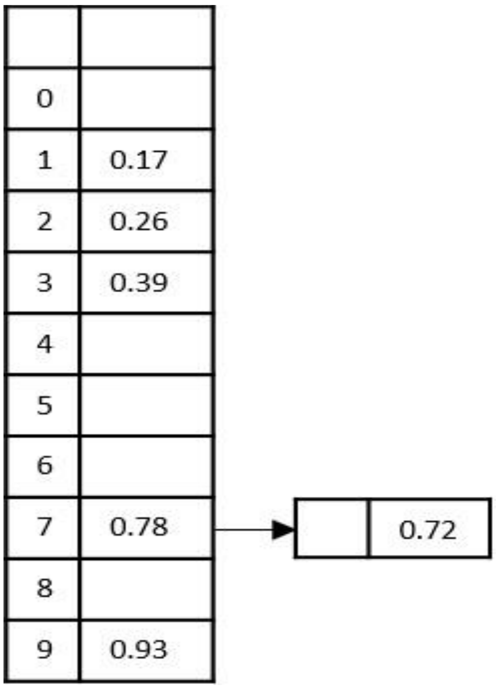


Figure 2.25

Step 7

Add the remaining numbers to the buckets in the similar manner by creating linked lists from the desired buckets. But while inserting these elements as lists, we apply insertion sort, i.e., compare the two elements and add the minimum value at the front as shown below −

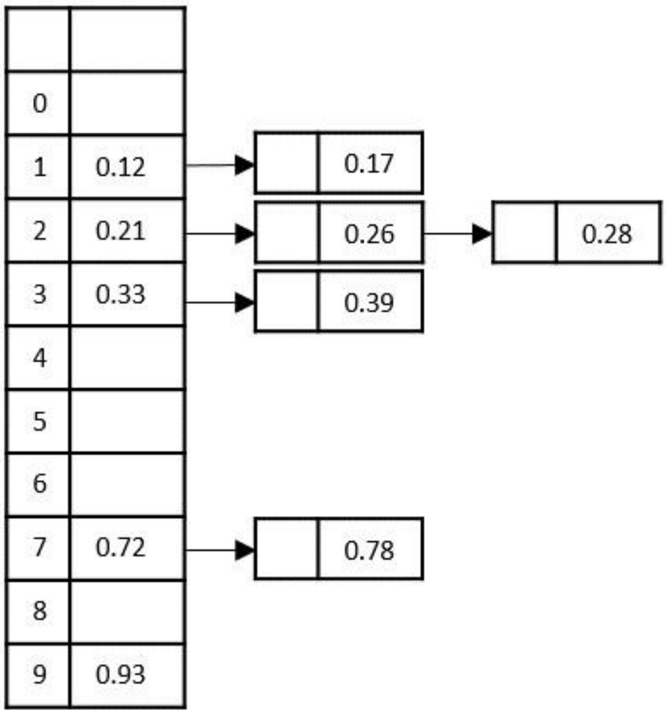


Figure 2.26

Step 8

Now, to achieve the output, concatenate all the buckets together.

0.12, 0.17, 0.21, 0.26, 0.28, 0.33, 0.39, 0.72, 0.78, 0.93

2.11.2 Bucket sort complexity

Here we discuss the time complexity of bucket sort in best case, average case, and in worst case and the space complexity of the bucket sort.

1. Time Complexity

- **Best Case Complexity –**

  It occurs when there is no sorting required, i.e. the array is already sorted. In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets. The complexity will be better if the elements are already sorted in the buckets.

If we use the insertion sort to sort the bucket elements, the overall complexity will be linear, i.e., O(n + k), where O(n) is for making the buckets, and O(k) is for sorting the bucket elements using algorithms with linear time complexity at best case. The best-case time complexity of bucket sort is **O(n + k)**.

- **Average Case Complexity –**

  It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. Bucket sort runs in the linear time, even when the elements are uniformly distributed. The average case time complexity of bucket sort is **O(n + K)**.

- **Worst Case Complexity –**

  In bucket sort, worst case occurs when the elements are of the close range in the array, because of that, they have to be placed in the same bucket. So, some buckets have more number of elements than others. The complexity will get worse when the elements are in the reverse order. The worst-case time complexity of bucket sort is **O(n^2)**.

2. Space Complexity

The space complexity of bucket sort is O(n*k).

SAMPLE QUESTIONS:

1. Solve the recurrence relation T(n) = 2T(n/2) + n, where T(1) = 1, using the substitution method.
2. Use the iterative method to solve the recurrence T(n) = 3T(n/2) + 1, with T(1) = 1.
3. Apply the recursion tree method to analyze the recurrence T(n) = T(n/3) + T(2n/3) +n.
4. Solve the recurrence relation T(n) = 4T(n/2) + n^2, where T(1) = 1, using the master method.
5. Prove the lower bound for comparison-based sorting algorithms is Ω(n log n) using a decision tree.
6. Implement the Counting Sort algorithm and demonstrate its application on an array of integers.
7. Explain the Radix Sort algorithm and its time complexity. Compare it with other sorting algorithms in terms of performance.
8. Describe the Bucket Sort algorithm and discuss its suitability for sorting floating-point numbers.

9. Analyze the best-case, average-case, and worst-case time complexity of the Bubble Sort algorithm.
10. Compare the time complexity of QuickSort and MergeSort in the average case. Explain your findings.

UNIT III: Brute Force and Divide-And-Conquer

3.1 Brute Force Algorithm

Brute force is a straightforward, exhaustive algorithmic technique that involves systematically trying all possible solutions to a problem until the correct one is found. It is a general problem-solving approach that can be applied to a wide range of problems, but it is not always the most efficient method, especially for large-scale or complex problems.

How a brute force algorithm works:

1. **Generate all possible solutions:** The algorithm generates all possible solutions to the problem. This often involves trying every combination of elements or values within a given range.
2. **Test each solution:** For each generated solution, the algorithm tests whether it satisfies the conditions of the problem or meets the desired criteria.
3. **Evaluate results:** If a solution is found that satisfies the problem's conditions, the algorithm stops and returns that solution. If no solution is found, the algorithm may return a special value indicating failure.

Advantages of a brute-force algorithm

- This algorithm finds all the possible solutions, and it also guarantees that it finds the correct solution to a problem.
- This type of algorithm is applicable to a wide range of domains.
- It is mainly used for solving simpler and small problems.
- It can be considered a comparison benchmark to solve a simple problem and does not require any particular domain knowledge.

Disadvantages of a brute-force algorithm

- It is an inefficient algorithm as it requires solving each and every state.
- It is a very slow algorithm to find the correct solution as it solves each state without considering whether the solution is feasible or not.
- The brute force algorithm is neither constructive nor creative as compared to other algorithms.

3.2 Travelling Salesman Problem:

The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution.

To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution.

Problem Statement

The problem states that "What is the shortest possible route that helps the salesman visits each city precisely once and returns to the origin city, provided the distances between each pair of cities given?"

Example

Consider City1, City2, City3, City4 and the distances between them as shown below. A salesman has to start from City 1, travel through all the cities and return to City1. There are various possible routes, but the problem is to find the shortest possible route.

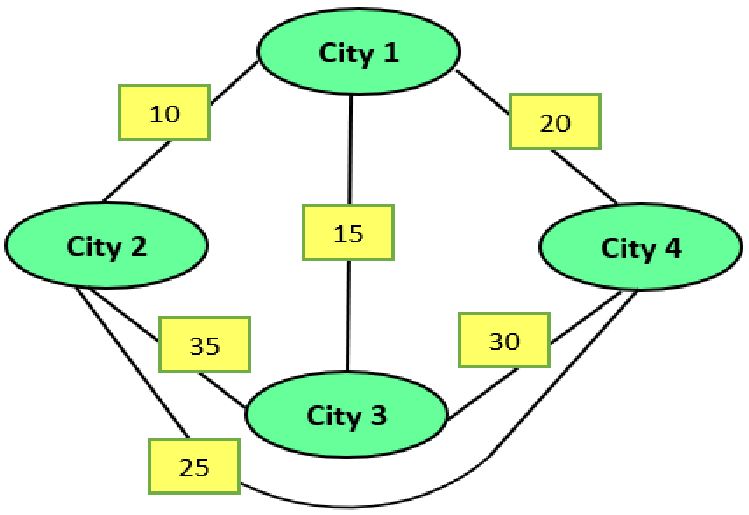


Figure 3.1

The various possible routes are:
Table 3.1.

Route	**Distance Travelled**
City(1 - 2 - 3 - 4 - 1)	95
City(1 - 2 - 4 - 3 - 1)	80
City(1 - 3 - 2 - 4 - 1)	95

Since we are travelling in a loop (source and destination are the same), paths like (1 - 2 - 3 - 4 - 1) and (1 - 4 - 3 - 2 - 1) are considered identical.

Out of the above three routes, route 2 is optimal. So, this will be the answer.

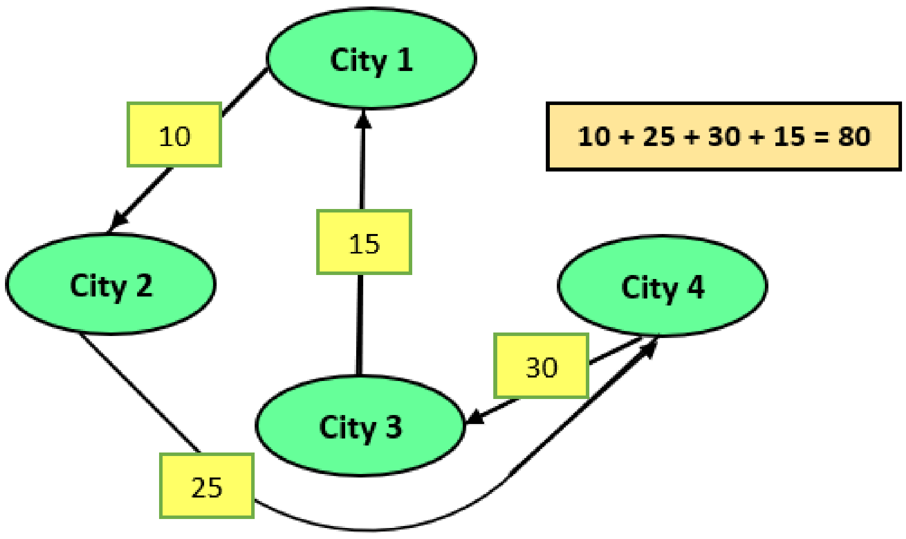


Figure 3.2

3.2.1 Solution Approach (Brute force)

There are various ways to solve this problem. But here we discuss the brute force approach. The travelling salesman problem is a permutation problem. There can be n! total ways to solve permutation problems.

In this approach, we'll generate all the possible permutations (routes) and find the one with the shortest distance.

The algorithm is as follows:

Step 1: Choose a city to act as a starting point (let's say city1).

Step 2: Find all possible routes using permutations. For n cities, it will be (n-1)!

Step 3: Calculate the cost of each route and return the one with the minimum cost.

3.2.2 Complexity Analysis

Time and Space Complexity:

The time complexity of **O(n^n)**, where n is the number of cities as we've to check (n-1)! routes (i.e all permutations).

The space complexity is **O(n^2)** as the n*n adjacency matrix needs O(n^2) space, and to store the remaining cities, we need O(n) space. The overall complexity will be O (n^2).

3.2.3 Application of Travelling Salesman Problem

The Travelling Salesman Problem (TSP) has numerous applications in various fields. Some of the common applications of TSP are:

1. **Logistics and transportation:** The TSP is used to optimize the routing of delivery trucks, sales representatives, and service technicians. By finding the shortest route that visits all necessary locations, TSP can help minimize travel time and reduce transportation costs.
2. **Circuit design:** In the field of electronics, TSP is used to optimize the routing of electrical signals in a printed circuit board, which is a critical step in the design of electronic circuits.
3. **DNA sequencing:** In bioinformatics, TSP is used to find the shortest path to sequence a DNA molecule. DNA sequencing involves finding the order of nucleotides in a DNA molecule, which can be modeled as a TSP.
4. **Network design:** TSP is used to optimize the design of communication networks, such as the placement of cell towers, routing of fiber-optic cables, and design of wireless mesh networks.
5. **Robotics:** TSP can be applied in robotics to optimize the path planning of robots to visit a set of locations in the most efficient manner.

These are just a few examples of the many applications of TSP. The problem is widely studied and has numerous practical applications in various fields.

3.3 Knapsack Problem

A knapsack problem algorithm is a constructive approach to combinatorial optimization. The problem is basically about a given set of items, each with a specific weight and a value. Therefore we needs to determine each item’s number to include in a knapsack so that the total weight is less than or equal to a given limit. And also, the total value is maximum.

The knapsack problem is a classic optimization problem where the goal is to maximize the total value of items included in a knapsack without exceeding its capacity. The brute-force method involves considering all possible combinations of items and selecting the one that satisfies the capacity constraint while maximizing the total value.

The Knapsack problem has two categories.

- 0-1 Knapsack Problem
- Fractional Knapsack Problem

3.3.1 0 - 1 Knapsack Problem

Let’s consider values and weights of n different items. One has to put these items in a knapsack of capacity C such that the knapsack has the maximum value. Also, the sum of weights of all the items present in the knapsack should not exceed the capacity C. Since it is a

0 - 1 knapsack problem; hence, splitting the item is not allowed, i.e., one can never break any given item, either do not pick it or pick it (0 - 1 property).

3.3.2 Fractional Knapsack Problem

The fractional knapsack problem is similar to the 0 - 1 knapsack problem. The only difference is one can choose an item partially. Thus, the liberty is given to break any item then put it in the knapsack, such that the total value of all the items (broken or not broken) present in the knapsack is maximized.

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

1. Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
2. Each item *i* has some weight *wi* and benefit value v*i*(all *wi* and *W* are integer values)
3. Problem: How to pack the knapsack to achieve maximum total value of packed items?

Problem, in other words, is to find

$$\max \sum_{i \in S} v_i \text{ subject to } \sum_{i \in S} w_i \le W$$

Given *n* items:

- weights: *w*1 *w2* ... *wn*
- values: *v*1 *v2* ... *vn*
- a knapsack of capacity *W*

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

Example 1: Knapsack capacity W=16

Table 3.2.

Item	**Weight**	**Value**
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10

Solution:

Table 3.3.

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

Example: 2

Given *n* = 3, (*p1, p2, p3*) = {25, 24, 15}
(w1, *w2, w3*) = {18, 15, 10} *M* = 20
Solution:

Some of the feasible solutions are shown in the following table.

Table 3.4.

Solution No	**x1**	**x2**	**x3**	**Σwi xi**	**Σpi xi**
1	1	2/15	0	20	28.2
2	0	2/3	1	20	31.0
3	0	1	1/2	20	31.5

The time complexity of the brute-force approach for the knapsack problem is exponential, specifically O(2^n), where n is the number of items. The reason for this high complexity lies in the nature of the brute-force method, which explores all possible combinations of items to find the optimal solution.

3.4 Assignment Problem

The assignment problem is another classic optimization problem where the goal is to assign a set of tasks to a set of agents in a way that minimizes the total cost or maximizes the total profit.

For example consider that there are n people and n jobs. Each person has to be assigned only one job. When the jth job is assigned to pth person the cost incurred is represented by C.

$$C=C[p,j]$$

Where, p=1,2,3..... n

J=1,2,3...... n

The number of permutations (the number of different assignments to different persons) is n! The exhaustive search is impractical for large value of n. Let us consider 4 persons (P1,P2,P3 and P4) and 4 jobs(J1,J2,J3 and J4).

Here n=4.Here the number of possible and different types of assignment is 4!

$$n! = 4!$$
$$=4 \times 3 \times 2 \times 1=24$$

The below table shows the entries representing the assignment costs C[p,j].

Table 3.5.

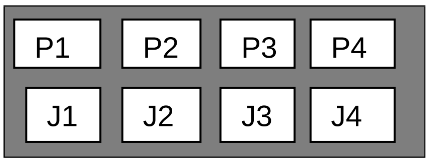
Job / Person	J1	J2	J3	J4
P 1	9	2	7	8
P2	6	4	3	7
P3	5	8	1	8
P4	7	6	9	4

Iterations of solving the above assignment problem are given below. Here 4 persons indicated by P1, P2, P3 and P4; Similarly 4 jobs are indicated by J1, J2, J3 and J4.

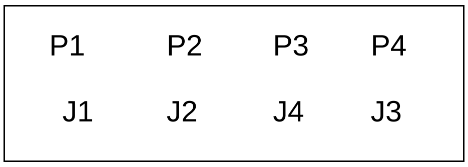
Let us consider that the assignments can be grouped into 4 groups.

In the first group J1 is assigned to person P1. The remaining jobs J2, J3 and J4 are assigned to persons P2, P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

Group-I



9+4+1+8 = 18



9+4+8+9 = 30

P1	P2	P3	P4
J1	J3	J2	J4

9+3+8+4= 24

P1	P2	P3	P4
J1	J3	J4	J2

9+3+8+6= 26

P1	P2	P3	P4
J1	J4	J2	J3

9+7+8+9= 33

P1	P2	P3	P4
J1	J4	J3	J2

9+7+1+6= 23

Group-2

In the second group J2 is assigned to person P1. The remaining jobs J3,J4,J1 are assigned to persons P2,P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J2	J3	J4	J1

2+3+8+7 = 20

P1	P2	P3	P4
J2	J1	J3	J4

2+6+1+4= 13

P1	P2	P3	P4
J2	J3	J1	J4

2+3+5+4 = 14

P1	P2	P3	P4
J2	J4	J3	J1

2+7+1+7= 17

P1	P2	P3	P4
J2	J4	J1	J3

2+7+5+9= 23

P1	P2	P3	P4
J2	J1	J4	J3

2+6+8+9= 25

Group-3

In the third group J3 is assigned to person P1. The remaining jobs J2,J4,J1 are assigned to persons P2,P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J3	J4	J1	J2

7+7+5+6 = 25

P1	P2	P3	P4
J3	J4	J2	J1

7+7+8+7 = 29

P1	P2	P3	P4
J3	J1	J4	J2

7+6+8+6= 27

P1	P2	P3	P4
J3	J2	J4	J1

7+4+8+7= 26

P1	P2	P3	P4
J3	J1	J2	J4

7+6+8+4= 25

P1	P2	P3	P4
J3	J2	J1	J4

7+4+5+4= 20

Group-4

In the Fourth group J4 is assigned to person P1. The remaining jobs J2, J3, J1 are assigned to persons P2, P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J4	J1	J2	J3

8+6+8+9 = 31

P1	P2	P3	P4
J4	J1	J3	J2

8+6+1+6 = 21

P1	P2	P3	P4
J4	J2	J1	J3

8+4+5+9= 26

P1	P2	P3	P4
J4	J2	J3	J1

8+4+1+7= 20

P1	P2	P3	P4
J4	J3	J1	J2

8+3+5+6= 22

P1	P2	P3	P4
J3	J3	J2	J1

8+3+8+7= 26

In the above four groups low costs are

Table 3.6

Group 1- 1st iteration is lowest	18
Group-II – 2nd iteration is lowest	13
Group-III- 6th iteration is lowest	20
Group-IV- 4th iteration is lowest	20

3.4.1 Complexity Analysis

The O(n! * n) time complexity is significant and makes the brute-force method impractical for larger instances of the assignment problem. The factorial growth in the number of permutations quickly becomes computationally infeasible as the number of agents or tasks increases.

3.5 Closest-Pair and Convex-Hull Problems

Let us consider a straight forward approach (Brute Force) to two well-known problems dealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

3.5.1 Closest-Pair Problem

The closest-pair problem finds 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.

Consider the two-dimensional case of the closest-pair problem. The points are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points pi(xi, yi) and pj(xj, yj) is the standard Euclidean distance.

Euclidean distance $d(P_i, P_j) = \sqrt{[(x_i-x_j)^2 + (y_i-y_j)^2]}$

The following algorithm computes the distance between each pair of distinct points and finds a pair with the smallest distance.

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 will be squaring a number. The number of times it will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2$$

$$= 2\sum_{j=i+1}^{n} (n-i)$$

$$= 2n(n-1)/2 \varepsilon \Theta(n^2)$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor, but it cannot improve its asymptotic efficiency class.

3.5.2 Convex-Hull Problem

3.5.2.1 Convex Set:

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.

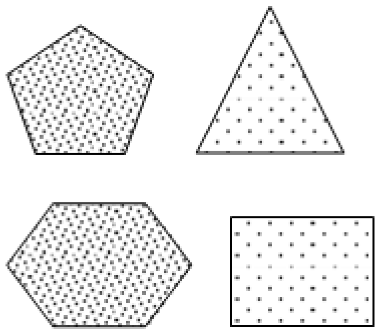


Figure 3.3 Convex sets.

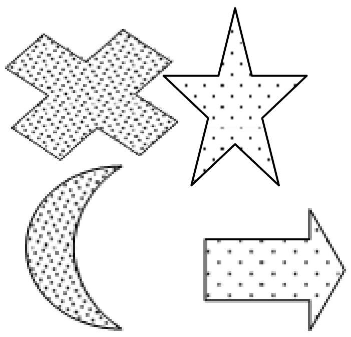


Figure 3.4 Not convex Sets.

All the sets depicted in Figure (3.3) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.

On the other hand, the sets depicted in Figure (3.4), 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.

Take a rubber band and stretch it to include all the nails, and then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in Figure (3.5).

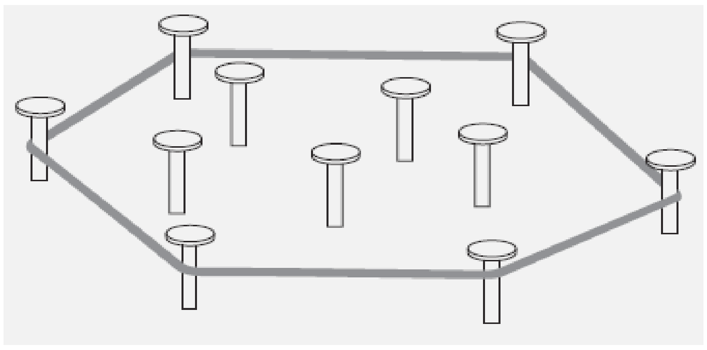


Figure 3.5 Rubber-band interpretation of the convex hull.

3.5.2.2 Convex hull

The ***convex hull*** of a set *S* of points is the smallest convex set containing *S*. (The smallest 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. For an example of the convex hull for a larger set, see Figure (3.6).

THEOREM

The convex hull of any set *S* of $n>2$ points not all on the same line is a convex polygon with the vertices at some of the points of *S*. (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of *S*.)

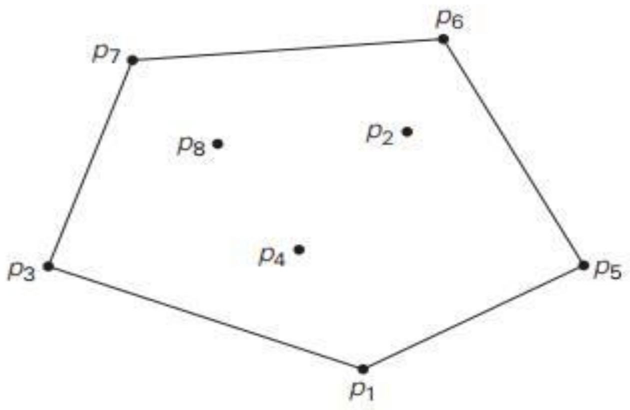


Figure 3.6 The Convex hull for this set of eight points is the convex polygon with vertices at p1, p5, p6, p7 and p3

The ***convex-hull problem*** is the problem of constructing the convex hull for a given set ***S*** of ***n*** points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon "extreme points." By definition, an ***extreme point*** of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 3.6 are p_1, p_5, p_6, p_7, and p_3.

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the ***simplex method***, a very important algorithm. This algorithm solves ***linear programming*** problems, which are problems of finding a minimum or a maximum of a linear function of $\boldsymbol{n}$ variables subject to linear constraints. We are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of $\boldsymbol{n}$ points of a given set extreme points of the set's convex hull are: we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counter clockwise order.

How can we solve the convex-hull problem in a brute-force manner? If you do not see an immediate plan for a frontal attack, do not be dismayed: the convex-hull problem is one with no obvious algorithmic solution. Nevertheless, there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$ of a set of $\boldsymbol{n}$ points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

A few elementary facts from analytical geometry are needed to implement this algorithm. First, the straight line through two points $\boldsymbol{(x_1, y_1), (x_2, y_2)}$ in the coordinate plane can be defined by the equation

$$ax + by = c,$$

where $a = y_2 - y_1,\ b = x_1 - x_2,\ c = x_1y_2 - y_1x_2.$

Second, such a line divides the plane into two half-planes: for all the points in one of them, $\boldsymbol{ax + by > c}$, while for all the points in the other, $\boldsymbol{ax + by < c}$. (For the points on the line itself, of course, $\boldsymbol{ax + by = c}$.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $\boldsymbol{ax + by - c}$ has the same sign for each of these points.

The time efficiency of this algorithm is $\boldsymbol{O(n^3)}$: for each of $\boldsymbol{n(n-1)/2}$ pairs of distinct points, we may need to find the sign of $\boldsymbol{ax + by - c}$ for each of the other $\boldsymbol{n} - 2$ points.

3.6 Divide and Conquer Approach

A **divide and conquer algorithm** works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**).

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

Divide And Conquer algorithm:

```
DAC(a, i, j)
{
   if(small(a, i, j))
     return(Solution(a, i, j))
   else
     mid = divide(a, i, j)            // f1(n)
     b = DAC(a, i, mid)               // T(n/2)
     c = DAC(a, mid+1, j)             // T(n/2)
     d = combine(b, c)                // f2(n)
  return(d)
}
```

Recurrence Relation for DAC Algorithm:

O(1) if n is small
T(n) =   f1(n) + 2T(n/2) + f2(n)

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple sub problems within the cache memory instead of accessing the slower main memory.

- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

3.7 Binary Search

Binary Search is a searching algorithm to find whether element x belongs to a set of numbers stored in an array. In each step, the algorithm compares the input element x with the value of the middle element in the array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for the left side of the middle element, else recurs for the right side of the middle element.

Algorithm BINSRCH (a, n, x)

```
// array a(1 : n) of elements in increasing order, n >= 0,
// determine whether 'x' is present, and if so, set j such that x = a(j)
// else return j
{
low :=1 ; high :=n ;
        while (low <= high) do
        {
        mid :=|(low + high)/2|
        if (x < a [mid]) then high:=mid – 1;
        else if (x > a [mid]) then low:= mid + 1
        else return mid;
        }
return 0;
}
```

low and high are integer variables such that each time through the loop either 'x' is found or low is increased by at least one or high is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually low will become greater than high causing termination in a finite number of steps if ‘x’ is not present.

Example: Binary Search

Let us illustrate binary search on the following 9 elements

Table 3.7.

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	**high**	**mid**
1	9	5
6	9	7
8	9	8
9	9	9

Found the element 101 and the No. of Comparisons = 4

Similarly we have to follow the procedure and find the elements. The number of element comparisons needed to find each of nine elements is:

Index:	1	2	3	4	5	6	7	8	9
Elements:	-15	-6	0	7	9	23	54	82	101
Comparisons:	3	2	3	4	1	3	2	3	4

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches	un-successful searches
Θ(1), Θ(log n), Θ(log n)	Θ(log n)
Best average worst	best, average and worst

3.8 Quick Sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the pivot element. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition( ) makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.
- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The algorithm terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted. Choose the first element as the 'pivot'. So, pivot = x[low]

Algorithm QUICKSORT (low, high)

```
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 :n) into
ascending order a (n + 1) is considered to be defined and must be greater than all elements in
a(1 : n); */
{
    if low < high then
    {
    j := PARTITION(a, low, high);
    // J is the position of the partitioning element
    QUICKSORT(low, j – 1);
    QUICKSORT(j + 1 , high);
    }
}
```

Algorithm PARTITION(a, m, p)

```
{
    V := a(m); i := m; j :=p; // A (m) is the partition element
    do
    {
        loop i := i + 1 until a(i) > v // i moves left to right
        loop j := j – 1 until a(j) < v // p moves right to left
        if (i < j) then INTERCHANGE(a, i, j)
    } while (i > j);
    a[m] :=a[j];
    a[j] :=V; // the partition element belongs at position P
    return j;
}
```

Algorithm INTERCHANGE(a, i, j)

```
{
    P:=a[i];
    a[i] := a[j];
    a[j] := p;
}
```

Example of Quick Sort:

44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than **44** then swap them. **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than 44 respectively.

Now we get two sorted lists:

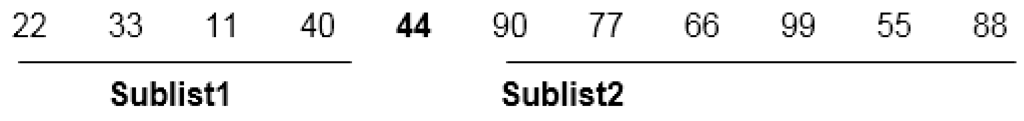


Figure 3.7

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

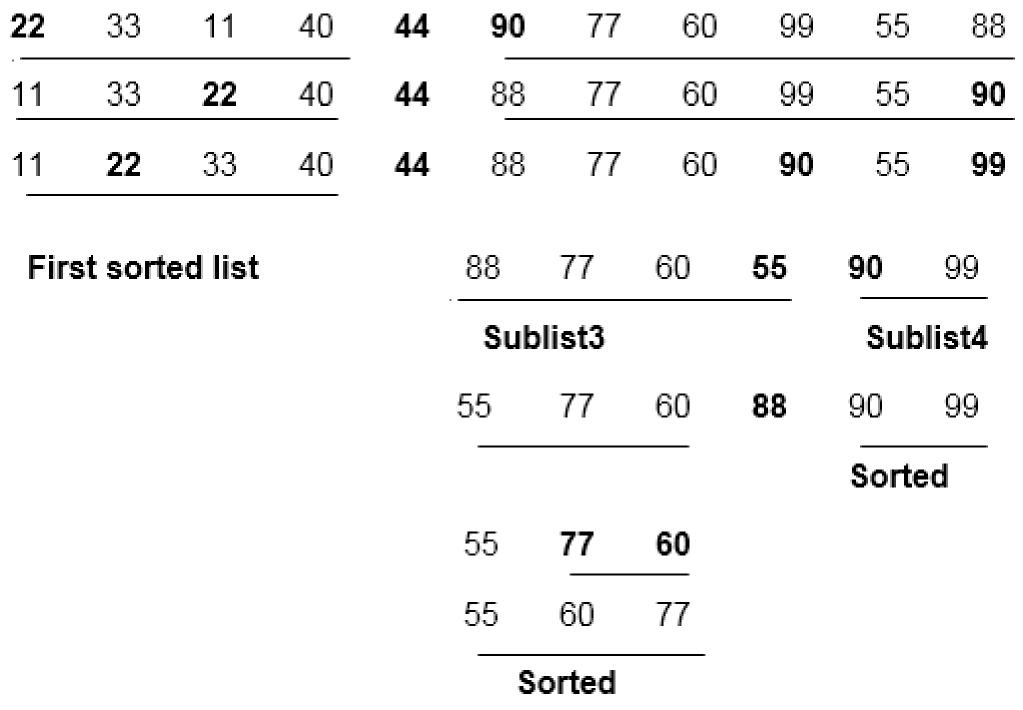


Figure 3.8

Merging Sublists:

11 22 33 40 44 55 60 77 88 90 99

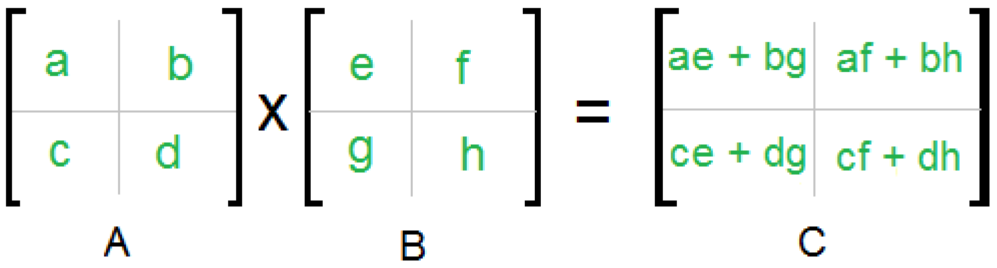
Figure 3.9 Sorted List

Analysis of Quick Sort:
Worst Case : O (n2)
Best Case : O(n log n)
Average Case : O(n log n)

3.9 Strassen’s Matrix Multiplication:

Strassen’s is simple Divide and Conquer method to multiply two square matrices.

1. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2. Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Figure 3.10

Strassen's insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22}) (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11}) (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$.

Time complexity of above method is O(n^2.81).

Problem: Multiply given two matrices A and B using Strassen's approach, where

Let's consider two matrices A and B:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Figure 3.11

Now, let's use Strassen's algorithm to find the product AB. The algorithm involves dividing the matrices into submatrices and performing seven multiplications recursively. The following steps outline the process:

Divide matrices A and B into four equal-sized submatrices:

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

$$B = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

Figure 3.12

Where A11,A12,A21,A22,B11,B12,B21,B22A11,A12,A21,A22,B11,B12,B21,B22 are submatrices.

Calculate the following products:

$$\begin{aligned}
P1 &= A11 \cdot (B12 - B22) \\
P2 &= (A11 + A12) \cdot B22 \\
P3 &= (A21 + A22) \cdot B11 \\
P4 &= A22 \cdot (B21 - B11) \\
P5 &= (A11 + A22) \cdot (B11 + B22) \\
P6 &= (A12 - A22) \cdot (B21 + B22) \\
P7 &= (A11 - A21) \cdot (B11 + B12)
\end{aligned}$$

Figure 3.13

Compute the submatrices of the result matrix C:

$$\begin{aligned}C11 &= P5 + P4 - P2 + P6\\ C12 &= P1 + P2\\ C21 &= P3 + P4\\ C22 &= P5 + P1 - P3 - P7\end{aligned}$$

Figure 3.14

Now, let's substitute the values and calculate the product AB:

$$\begin{aligned}C11 &= (47 + 48) + (0 - 0) - (42 + 56) + (18 - 0) = 55\\ C12 &= (0 + 0) + (47 + 48) = 95\\ C21 &= (0 + 0) + (0 - 0) = 0\\ C22 &= (47 + 48) + (0 + 0) - (0 + 0) - (0 + 0) = 95\end{aligned}$$

Figure 3.15

So, the product of matrices A and B using Strassen's algorithm is:

$$AB = \begin{bmatrix} 55 & 95 \\ 0 & 95 \end{bmatrix}$$

Figure 3.16

3.10 Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each sub array, and then merging the sorted subarrays back together to form the final sorted array.

It is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.
The Best and worst case time complexity of Merge sort is O(n log n).

Merge Sort Algorithm

Step 1 − if it is only one element in the list, consider it already sorted, so return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Algorithm **MERGESORT** (low, high)

```
// a (low : high) is a global array to be sorted.
{
        if (low < high)
        {
        mid := |(low + high)/2| //finds where to split the set
        MERGESORT(low, mid) //sort one subset
        MERGESORT(mid+1, high) //sort the other subset
        MERGE(low, mid, high) // combine the results
        }
}
```

Algorithm **MERGE** (low, mid, high)

```
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        h :=low; i := low; j:= mid + 1;
        while ((h < mid) and (J < high)) do
        {
                if (a[h] < a[j]) then
                {
                b[i] := a[h]; h := h + 1;
                }
                else
                {
                b[i] :=a[j]; j := j + 1;
                }
        i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
        }
        else
        for k := h to mid do
        {
            b[i] := a[K]; i := i + l;
        }
    for k := low to high do
    a[k] := b[k];
}
```

Example:

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.

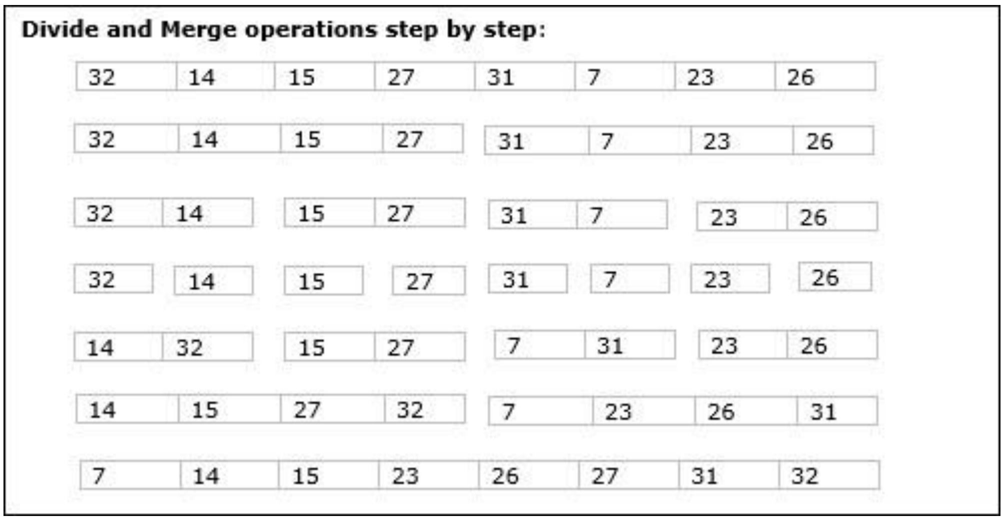


Figure 3.17

SAMPLE QUESTIONS:

1. Write the concept of brute force approach. Give one example.
2. Solve the Traveling Salesman Problem for a given set of cities with their respective distances. Discuss the time complexity of your solution.
3. Implement the 0/1 Knapsack Problem using a Brute Force approach. Provide the optimal subset and its total value.
4. Formulate an assignment problem and solve it using the Brute Force approach. Show the step-by-step assignment process.

5. Given a set of points in a plane, implement an algorithm to find the closest pair using the divide and conquer approach.
6. Discuss the Convex Hull problem with the concept of brute force approach
7. Apply the divide and conquer approach to find the maximum subarray sum in a given array. Provide the algorithm and analyze its time complexity.
8. Write an iterative binary search algorithm to find the index of a specific element in a sorted array. Discuss the time complexity of your solution.
9. Implement the Quick Sort algorithm and demonstrate its application on an array of integers. Discuss the partitioning process at each step.
10. Apply Strassen's algorithm to multiply two 2x2 matrices. Show the step-by-step process and analyze the time complexity.
11. Implement the Merge Sort algorithm and compare its time complexity with other sorting algorithms. Discuss its advantages and disadvantages.

UNIT IV: Greedy Approach and Dynamic Programming

4.1 Greedy Approach:

Greedy method is the simplest and straightforward approach. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

4.1.1 Components of Greedy Algorithm

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

4.1.2 Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

```
Algorithm Greedy (a, n)
{
        Solution : = 0;
                for i = 0 to n do
                {
                  x: = select(a);
                        if feasible(solution, x)
                        {
                        Solution: = union(solution , x)
                        }
                        return solution;
                 }
}
```

Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

4.2 Optimal Merge Patterns

Given n sorted files, find an optimal way (i.e., requiring the fewest comparisons or record moves) to pair wise merge them into one sorted file. It fits ordering paradigm.

Example:

Three sorted files (x1, x2, x3) with lengths (30, 20, 10)

Solution 1: merging x1 and x2 (50 record moves), merging the result with x3 (60 moves)

Totally 110 moves

Solution 2: merging x2 and x3 (30 moves), merging the result with x1 (60 moves)

Totally 90 moves

The solution 2 is better.

"Merge n sorted sequences of different lengths into one sequence while minimizing reads". Any two sequences can be merged at a time. At each step, the two shortest sequences are merged.

Consider three sorted lists L_1, L_2 and L_3 of size 30, 20 and 10 respectively. Two way merge compares elements of two sorted lists and put them in new sorted list.

If we first merge list L_1 and L_2, it does 30 + 20 = 50 comparisons and creates a new array L' of size 50. L' and L_3 can be merged with 60 + 10 = 70 comparisons that forms a sorted list L of size 60. Thus total number of comparisons required to merge lists L_1, L_2 and L_3 would be 50 + 70 = 120.

Alternatively, first merging L_2 and L_3 does 20 + 10 = 30 comparisons, which creates sorted list L' of size 30. Now merge L_1 and L', which does 30 + 30 = 60 comparisons to form a final sorted list L of size 60. Total comparisons required to create L is thus 30 + 60 = 90.

In both the cases, final output is identical but first approach does 120 comparisons whereas second does only 90. Goal of optimal merge pattern is to find the merging sequence which results into minimum number of comparisons.

Let S = $\{s_1, s_2, \ldots, s_n\}$ be the set of sequences to be merged. Greedy approach selects minimum length sequences s_i and s_j from S. The new set S' is defined as, S' = $(S - \{s_i, s_j\}) \cup \{s_i + s_j\}$. This procedure is repeated until only one sequence is left.

Complexity Analysis

The total running time of this algorithm is O(nlogn).

4.3 Huffman Code

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters; lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

(i) Data can be encoded efficiently using Huffman Codes.

(ii) It is a widely used and beneficial technique for compressing data.

(iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

Table 4.1.

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

a	000
b	001
c	010
d	011
e	100
f	101

For a file with 10^5 characters, we need 3 x 10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long code words.

For example:

a	0
b	101
c	100
d	111
e	1101
f	1100

Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000 **= 2.24 x 10^5bits**

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

4.3.1 Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the code word that starts with an encoded data is unambiguous.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

```
Algorithm Huffman (c)
{
  n= |c|
  Q = c
  for i<-1 to n-1
  do
  {
     temp <- get node ()
    left (temp] Get_min (Q) right [temp] Get Min (Q)
    a = left [templ b = right [temp]
    F [temp]<- f[a] + [b]
    insert (Q, temp)
   }
return Get_min (0)
}
```

4.3.2 Steps to build Huffman Tree:

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

4.4 Job Sequencing with Deadline

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

4.4.1 Job Scheduling Algorithm

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

Algorithm

- Find the maximum deadline value from the input set of jobs.
- Once, the deadline is decided, arrange the jobs in descending order of their profits.
- Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.
- The selected set of jobs is the output.

Examples

Consider the following tasks with their deadlines and profits. Schedule the tasks in such a way that they produce maximum profit after being executed .

Table 4.2.

S. No.	1	2	3	4	5
Jobs	J1	J2	J3	J4	J5
Deadlines	2	2	1	3	4
Profits	20	60	40	100	80

Step 1

Find the maximum deadline value, dm, from the deadlines given.

$d_m = 4$.

Step 2

Arrange the jobs in descending order of their profits.

Table 4.3.

S. No.	1	2	3	4	5
Jobs	J4	J5	J2	J3	J1

Deadlines	3	4	2	1	2
Profits	100	80	60	40	20

The maximum deadline, d_m, is 4. Therefore, all the tasks must end before 4.

Choose the job with highest profit, J4. It takes up 3 parts of the maximum deadline.

Therefore, the next job must have the time period 1.

Total Profit = 100.

Step 3

The next job with highest profit is J5. But the time taken by J5 is 4, which exceeds the deadline by 3. Therefore, it cannot be added to the output set.

Step 4

The next job with highest profit is J2. The time taken by J5 is 2, which also exceeds the deadline by 1. Therefore, it cannot be added to the output set.

Step 5

The next job with higher profit is J3. The time taken by J3 is 1, which does not exceed the given deadline. Therefore, J3 is added to the output set.

Total Profit: 100 + 40 = 140

Step 6

Since, the maximum deadline is met, the algorithm comes to an end. The output set of jobs scheduled within the deadline are **{J4, J3}** with the maximum profit of **140**.

Example

Let n = 4, (p_1,p_2,p_3,p_4) = (100,10,15,27), (d_1,d_2,d_3,d_4) = (2,1,2,1)

Table 4.4.

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1, 2)	(2, 1)	110
(ii)	(1, 3)	(1, 3) or (3, 1)	115
(iii)	(1, 4)	(4, 1)	127 is the optimal one
(iv)	(2, 3)	(2, 3)	25
(v)	(3, 4)	(4, 3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

4.5 Tree Vertex Splitting

- Directed and weighted binary tree.
- Consider a network of power line transmission.
- The transmission of power from one node to the other results in some loss, such as drop in voltage.
- Each edge is labeled with the loss that occurs (edge weight).
- Network may not be able to tolerate losses beyond a certain level.
- Place boosters in the nodes to account for the losses.

Definition

Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters. Place boosters only in the vertices and nowhere else.

More definitions

- Let T = (V, E, w) be a weighted directed tree
- V is the set of vertices
- E is the set of edges
- w is the weight function for the edges
- w_{ij} is the weight of the edge $\langle i, j\rangle \in E$
- We say that $w_{ij} = \infty$ if $\langle i, j\rangle \notin E$
- A vertex with in-degree zero is called a source vertex
- A vertex with out-degree zero is called a sink vertex
- For any path $P \in T$, its delay d(P) is defined to be the sum of the weights (w_{ij}) of that path, or

$$d(P) = \sum_{\langle i,j\rangle \in P} w_{ij}$$

- Delay of the tree T, d(T) is the maximum of all path delays
- Splitting vertices to create forest
    - ✓ Let T /X be the forest that results when each vertex $u \in X$ is split into two nodes u^i and u^o such that all the edges $\langle u, j\rangle \in E$ [$\langle j, u\rangle \in E$] are replaced by edges of the form

      $\langle u^o, j\rangle \in E$ [$\langle j, u^i\rangle \in E$]
        - Outbound edges from u now leave from u^o
        - Inbound edges to u now enter at u^i

- ✓ Split node is the booster station
- ➢ Tree vertex splitting problem is to identify a set X ⊆ V of minimum cardinality (minimum number of booster stations) for which $d(T/X) \le \delta$ for some specified tolerance limit δ
  - o TVSP has a solution only if the maximum edge weight is ≤ δ
- ➢ Given a weighted tree T = (V, E, w) and a tolerance limit δ, any X ⊆ V is a feasible solution if $d(T/X) \le \delta$
  - ✓ Given an X, we can compute d(T /X) in O(|V |) time
  - ✓ A trivial way of solving TVSP is to compute d(T /X) for every X ⊆ V, leading to a possible $2^{|V|}$ computations

Greedy solution for TVSP

– We want to minimize the number of booster stations (X)

– For each node u ∈ V, compute the maximum delay d(u) from u to any other node in its subtree

– If u has a parent v such that $d(u) + w(v, u) > \delta$, split u and set d(u) to zero

– Computation proceeds from leaves to root

– Delay for each leaf node is zero

– The delay for each node v is computed from the delay for the set of its children C(v)

$$d(v) = \max_{u \in C(v)} \{d(u) + w(v, u)\}$$

If $d(v) > \delta$, split v

– The above algorithm computes the delay by visiting each node using post-order traversal.

```
int TVS ( tree T, int delta )
{
        if ( T == NULL ) return ( 0 ); // Leaf node
        d_l = tvs ( T.left(), delta ); // Delay in left subtree
        d_r = tvs ( T.right(), delta ); // Delay in right subtree
        current_delay = max ( w_l + d_l,// Weight of left edge
                          w_r + d_r ); // Weight of right edge
        if ( current_delay > delta )
        {
                if ( w_l + d_l > delta )
                {
                        write ( T.left().info() );
                        d_l = 0;
                }
                if ( w_r + d_r > delta )
                {
                        write ( T.right().info() );
                        d_r = 0;
                }
        }
        current_delay = max ( w_l + d_l, w_r + d_r );
        return ( current_delay );
}
```

Algorithm TVS runs in Θ(n) time

∗ TVS is called only once on each node in the tree

∗ On each node, only a constant number of operations are performed, excluding the time for recursive calls

Example:

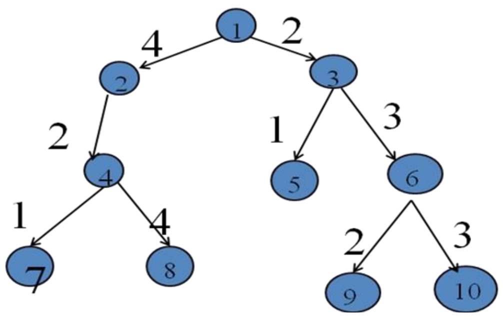


Figure 4.1

If d (u)>= δ than place the booster.

d (7)= max{0+w(4,7)}=1

d (8)=max{0+w(4,8)}=4

d (9)= max{0+ w(6,9)}=2

d (10)= max{0+w(6,10)}=3 d(5)=max{0+e(3.3)}=1

d (4)= max{1+w(2,4), 4+w(2,4)}=max{1+2,4+3}=6> δ ->booster

d (6)=max{2+w(3,6),3+w(3,6)}=max{2+3,3+3}=6> δ->booster

d (2)=max{6+w(1,2)}=max{6+4)=10> δ->booster d (3)=max{1+w(1,3), 6+w(1,3)}=max{3,8}=8> δ ->booster

Note: No need to find tolerance value for node 1 because from source only power is transmitting.

4.6 Dynamic Programming

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively.

The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these subproblems.

The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems**:

  The main problem is divided into smaller subproblems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.
- **Storing the sub solutions in a table**:

  The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.
- **Bottom-up computation**:

  The DP technique starts with the smallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

The strategy can be used when the process of obtaining a solution of a problem can be viewed as a sequence of decisions. The problems of this type can be solved by taking an optimal sequence of decisions. An optimal sequence of decisions is found by taking one decision at a time and never making an erroneous decision. In Dynamic Programming, an optimal sequence of decisions is arrived at by using the principle of optimality. The principle of optimality states that whatever be the initial state and decision, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting form the first decision.

A fundamental difference between the greedy strategy and dynamic programming is that in the greedy strategy only one decision sequence is generated, wherever in the dynamic programming, a number of them may be generated. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

4.7 Dice Throw

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.

The **Naive approach** is to find all the possible combinations of values from n dice and keep on counting the results that sum to X.

This problem can be efficiently solved using **Dynamic Programming (DP)**.

Let the function to find X from n dice is: Sum(m, n, X)

The function can be represented as:

```
Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth dice
             + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth dice
             + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth dice
               ..................................................
               ..................................................
                ..................................................
             + Finding Sum (X - m) from (n - 1) dice plus m from nth dice
```

So we can recursively write Sum(m, n, x) as following

```
        Sum(m, n, X) = Sum(m, n - 1, X- 1) +
                       Sum(m, n - 1, X - 2) +
                       .................... +
                       Sum(m, n - 1, X - m)
```

4.7.1 Why DP approach?

The above problem exhibits overlapping subproblems. See the below diagram. Also, see this recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:

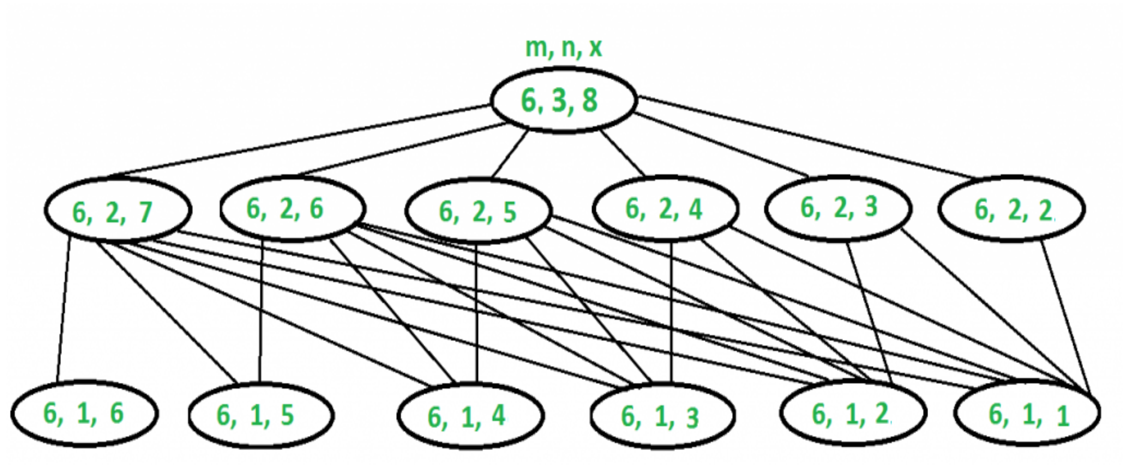


Figure 4.2

Sum(6, 3, 8) = Sum(6, 2, 7) + Sum(6, 2, 6) + Sum(6, 2, 5) + Sum(6, 2, 4) + Sum(6, 2, 3) + Sum(6, 2, 2)

To evaluate Sum(6, 3, 8), we need to evaluate Sum(6, 2, 7) which can recursively written as following:

Sum(6, 2, 7) = Sum(6, 1, 6) + Sum(6, 1, 5) + Sum(6, 1, 4) + Sum(6, 1, 3) + Sum(6,1, 2) + Sum(6, 1, 1)

We also need to evaluate Sum(6, 2, 6) which can recursively written as following:

Sum(6, 2, 6) = Sum(6, 1, 5) + Sum(6, 1, 4) + Sum(6, 1, 3) + Sum(6, 1, 2) + Sum(6,1, 1)

....

....

Sum(6, 2, 2) = Sum(6, 1, 1)

Time Complexity: O(m * n * x) where m is number of faces, n is number of dice and x is given sum.

Auxiliary Space: O(n * x)

4.8 Optimal Binary Search Tree (OBST)

An Optimal Binary Search Tree (OBST) is a specialized version that minimizes the cost of search operations, making it highly efficient. An Optimal Binary Search Tree is a variant of binary search trees where the arrangement of nodes is strategically optimized to minimize the cost of searches. The key idea behind an OBST is to place frequently accessed items closer to the root, reducing the search time for those elements. Conversely, less frequently accessed items are placed further away, leading to a balanced and efficient search tree.

The dynamic approach to constructing Optimal Binary Search Trees (OBSTs) is a powerful technique that utilizes dynamic programming to find the optimal solution efficiently. The fundamental idea behind dynamic programming is to break down a complex problem into smaller, overlapping subproblems and solve each subproblem only once, storing its solution for future reference. This "memoization" technique eliminates redundant calculations and significantly speeds up the overall process.

4.8.1 Optimal Substructure

A key concept in the dynamic programming approach is "optimal substructure." It means that the optimal solution to a larger problem can be obtained by combining the optimal solutions of its smaller subproblems. For OBSTs, the optimal substructure property is crucial because it allows us to find the best arrangement of nodes in a subtree and then reuse this solution when constructing the overall tree.

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2). Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.

The OBST is a useful data structure in applications where the keys have different probabilities of being searched for. It can be used to improve the efficiency of searching and retrieval operations in databases, compilers, and other computer programs.

Given a sorted array key [0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches for keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

Example:

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs

```
        10              12
          \            /
           12         10
          I            II
```

Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is 34*1 + 50*2 = 134

The cost of tree II is 50*1 + 34*2 = 118

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs

```
    10           12            20      10          20
      \         /  \          /          \        /
      12      10    20       12           20     10
        \                   /            /         \
        20                 10           12          12

      I          II           III         IV          V
```

Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142

4.8.2 Optimal Substructure:

The optimal cost for freq[i..j] can be recursively calculated using the following formula.

We need to calculate ***optCost(0, n-1)*** to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.

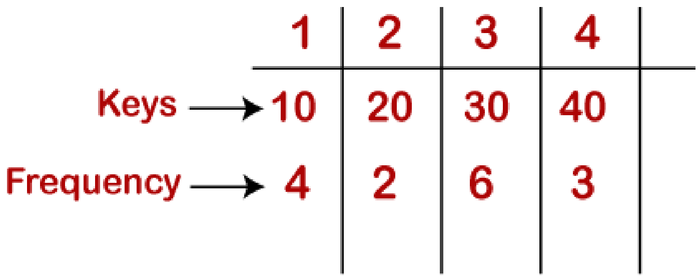
We add sum of frequencies from i to j (see first term in the above formula)

The reason for adding the sum of frequencies from i to j:

This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements from i to j except r. The term freq[r] is added because it is going to be root and that means level of 1, so freq[r]*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining elements because as we take r as root then all the elements other than that are going 1 level down than that is calculated in the subproblem. Let me put it in a more clear way, for calculating optcost(i,j) we assume that the r is taken as root and calculate min of opt(i,r-1)+opt(r+1,j) for all i<=r<=j. Here for every subproblem we are choosing one node as a root. But in reality the level of subproblem root and all its descendant nodes will be 1 greater than the level of the parent problem root. Therefore the frequency of all the nodes except r should be added which accounts to the descend in their level compared to level assumed in subproblem.

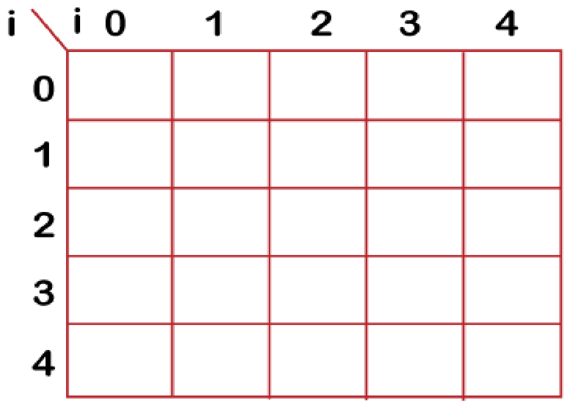
4.8.3 Dynamic Approach

Consider the below table, which contains the keys and frequencies.



	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3

Figure 4.3



i \ i	0	1	2	3	4
0					
1					
2					
3					
4					

Figure 4.4

First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

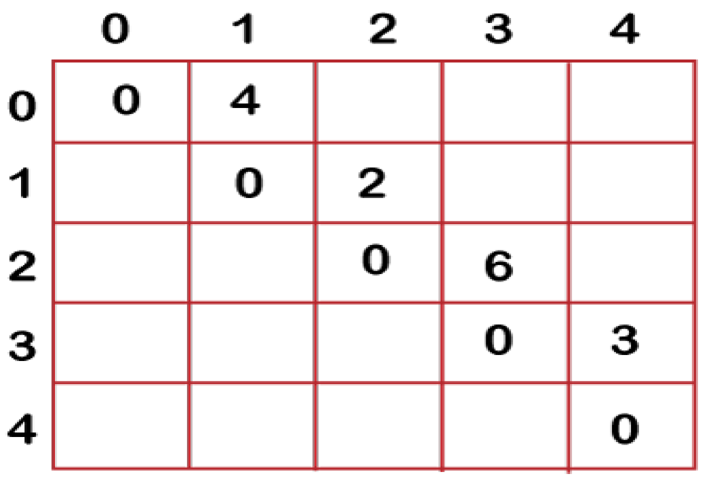
Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)

The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)



	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Figure 4.5

Now we will calculate the values where j-i = 2

When j=2, i=0 then j-i = 2

When j=3, i=1 then j-i = 2

When j=4, i=2 then j-i = 2

In this case, we will consider two keys.

When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:

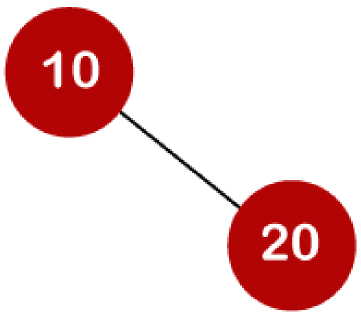
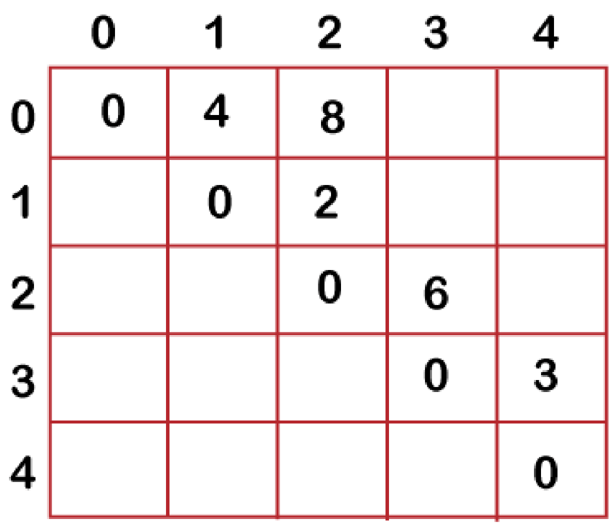


Figure 4.6

In the first binary tree, cost would be: 4*1 + 2*2 = 8

In the second binary tree, cost would be: 4*2 + 2*1 = 10

The minimum cost is 8; therefore, c[0,2] = 8



	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

Figure 4.7

- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

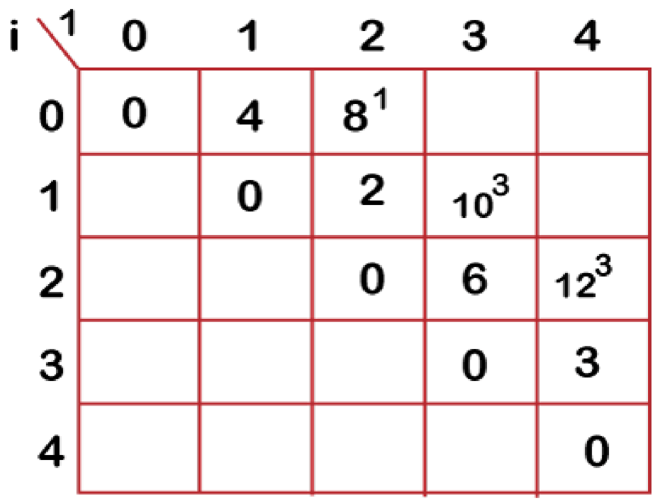
In the first binary tree, cost would be: 1*2 + 2*6 = 14

In the second binary tree, cost would be: 1*6 + 2*2 = 10

The minimum cost is 10; therefore, c[1,3] = 10

- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:
- In the first binary tree, cost would be: 1*6 + 2*3 = 12

- In the second binary tree, cost would be: 1*3 + 2*6 = 15
- The minimum cost is 12, therefore, c[2,4] = 12



i \ 1	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

Figure 4.8

Now we will calculate the values when j-i = 3

When j=3, i=0 then j-i = 3

When j=4, i=1 then j-i = 3

- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.

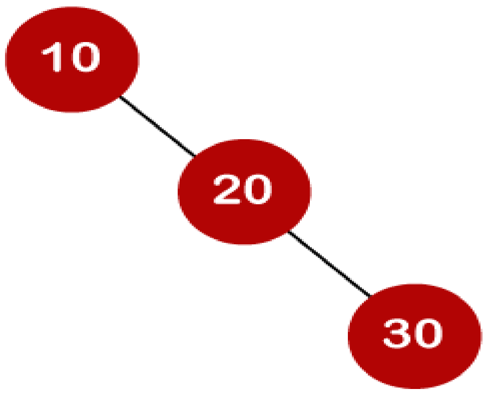


Figure 4.9

In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

Cost would be: 1*4 + 2*2 + 3*6 = 26

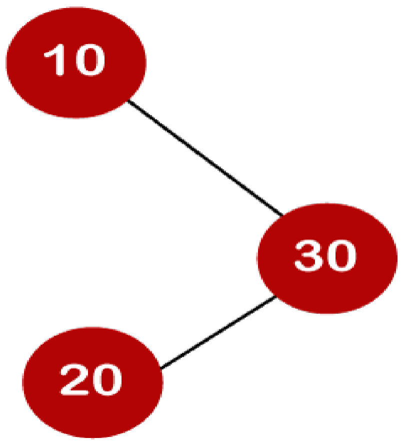


Figure 4.10

In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: 1*4 + 2*6 + 3*2 = 22

The following tree can be created if 20 is considered as the root node.

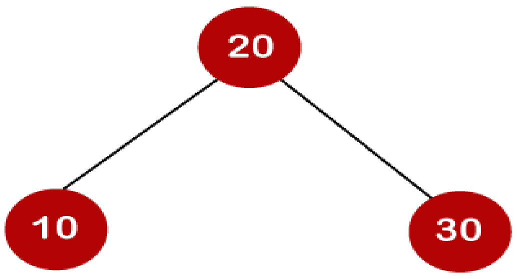


Figure 4.11

In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

Cost would be: 1*2 + 4*2 + 6*2 = 22

The following are the trees that can be created if 30 is considered as the root node.

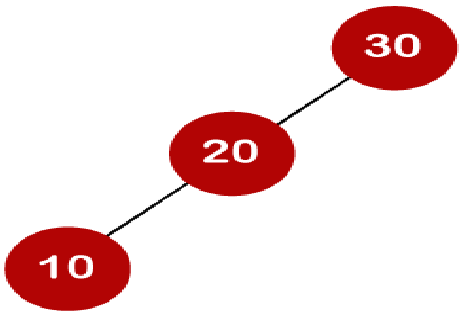


Figure 4.12

In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: 1*6 + 2*2 + 3*4 = 22

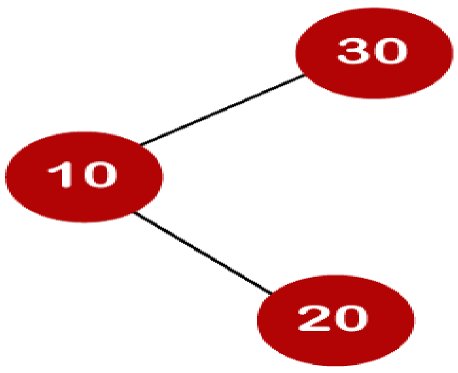


Figure 4.13

In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

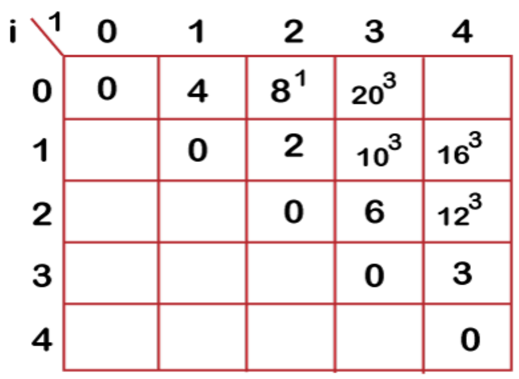
- When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

= min{0+12, 2+3, 10+0}+ 11

= min{12, 5, 10} + 11

The minimum value is 5; therefore, c[1,4] = 5+11 = 16



i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Figure 4.14

- **Now we will calculate the values when j-i = 4**

When j=4 and i=0 then j-i = 4

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then

C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]

= min {0 + 16} + 15= 31

If we consider 20 as the root node then

C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]

= min {8 + 3} + 15

= 26

If we consider 40 as the root node then,

C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]

= min{20 + 0} + 15

= 35

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Figure 4.15

The optimal binary tree can be created as:

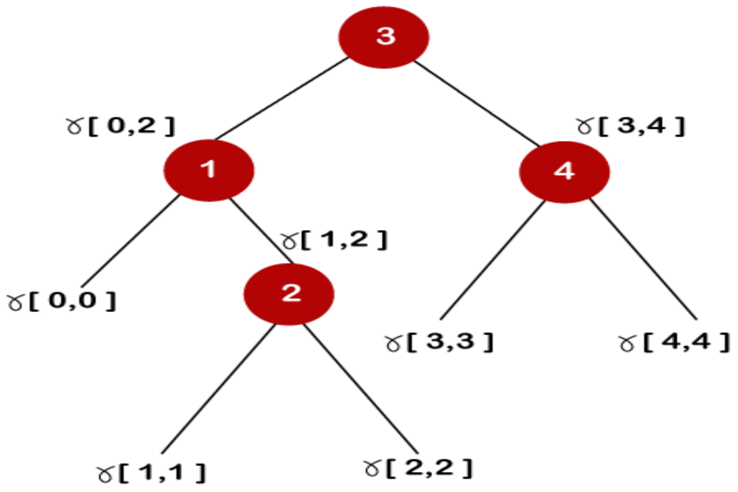


Figure 4.16

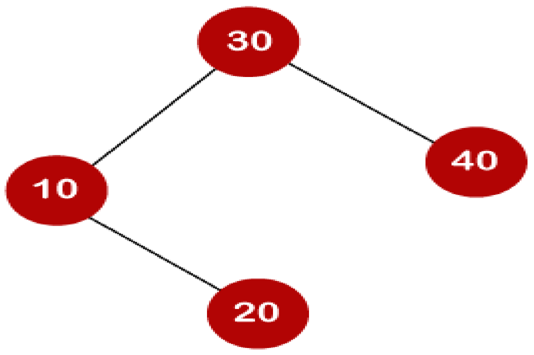


Figure 4.17

General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

SAMPLE QUESTIONS:

1. Given a set of files with different sizes, implement an algorithm to find the optimal way to merge them using the greedy approach.
2. Construct a Huffman tree for a set of characters with their respective frequencies. Encode and decode a sample string using the generated Huffman codes.
3. Solve a job sequencing problem with deadlines and profits using the greedy approach. Show the order of job execution to maximize profits.
4. Implement an algorithm to split a tree into two subtrees to minimize the maximum height difference between them.
5. Given a dice with 'm' faces and 'n' throws, implement a dynamic programming algorithm to find the number of ways to get a specific sum 'x.'
6. Given a set of keys and their probabilities, construct an optimal binary search tree using dynamic programming. Calculate the expected search cost.

UNIT V: Backtracking and Branch and Bound

5.1 Back Tracking:

In back tracking technique, solve the problems in an efficient way, when compared to other methods like greedy method and dynamic programming. The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P(x1, …. xn). Form a solution at any point seems not promising, ignore it.

All possible solutions require a set of constraints divided into two categories:

1. **Explicit Constraint:** Explicit constraints are rules that restrict each xi to take on values only from a given set. Examples x_i>= 0 or x1= 0 or 1 or l_i<=x_i<=u_i.

2. Implicit Constraint: Implicit Constraints are rules that determine which of the tuples in the solutions space of I satisfy the criterion function.

Back tracking is a modified depth first search tree. Backtracking is a procedure whereby, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child. State space tree exists implicitly in the algorithm because it is not actually constructed.

5.1.1 Terminologies which is used in this method:

- **Solution space**: Tuples that satisfy the explicit constraints define a **solution space**.
- **Problem state**: The solution space can be organized into a tree. Each node in the tree defines a **problem state**.
- **State-space:** All paths from the root to other nodes define the **state-space** of the problem.
- **Solution states: Solution states** are those states leading to a tuple in the solution space.
- **Answer nodes** are those solution states leading to an answer-tuple (i.e. tuples which satisfy implicit constraints).
- **LIVE NODE:** A node which has been generated and all of whose children are not yet been generated.
- **E-NODE (Node being expanded)**: The live node whose children are currently being generated.
- **DEAD NODE**: A node that is either not to be expanded further, or for which all of its children have been generated

- **DEPTH FIRST NODE GENERATION:** In this, as soon as a new child C of the current E-node R is generated, C will become the new E-node.

```
ALGORITHM Back Tracking ($v_1,...,v_i$)
{
        IF ($v_1,...,v_i$) is a solution THEN RETURN ($v_1,...,v_i$)
                FOR each v DO
                        IF ($v_1,...,v_i$,v) is acceptable vector THEN
                        sol = try($v_1,...,v_i$,v)
                IF sol != ( ) THEN RETURN sol
             END
        END
RETURN ( )
}
```

If S_i is the domain of v_i, then $S_1 \times ... \times S_m$ is the solution space of the problem. The validity criteria used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

5.2 8-Queens problem

A classic combinatorial problem is to place 8 queens on a 8*8 chess board so that no two attack, i.,e no two queens are to the same row, column or diagonal. The algorithm of 8 queens problem can be obtained by placing n=8, in N queens algorithm.

We observe that, for every element on the same diagonal which runs from the upper left to the lower right, each element has the same "row-column" value. Also every element on the same diagonal which goes from upper right to lower left has the same "row+column" value.

If two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if

$i-j=k-l$ ……………….(1) or

$i+j=k+l$ ……………….(2).

From (1) and (2) implies

$j-l=i-k$ and

$j-l=k-i$

Two queens lie on the same diagonal iff

$|j-l|=|i-k|$

5.2.1 Backtracking Approach

This approach rejects all further moves if the solution is declined at any step, goes back to the previous step and explores other options.

Algorithm

Let's go through the steps below to understand how this algorithm of solving the 8 queens problem using backtracking works:

- **Step 1:** Traverse all the rows in one column at a time and try to place the queen in that position.
- **Step 2:** After coming to a new square in the left column, traverse to its left horizontal direction to see if any queen is already placed in that row or not. If a queen is found, then move to other rows to search for a possible position for the queen.
- **Step 3:** Like step 2, check the upper and lower left diagonals. We do not check the right side because it's impossible to find a queen on that side of the board yet.
- **Step 4:** If the process succeeds, i.e. a queen is not found, mark the position as '1' and move ahead.
- **Step 5:** Recursively use the above-listed steps to reach the last column. Print the solution matrix if a queen is successfully placed in the last column.
- **Step 6:** Backtrack to find other solutions after printing one possible solution.

5.2.2 Complexity Analysis

Time Complexity: O(N!), For the first column we will have N choices, then for the next column, we will have N-1 choices, and so on. Therefore the total time taken will be N*(N-1)*(N-2)...., which makes the time complexity to be O(N!).

Space Complexity: O(N^2), we can see that the initial space needed is N^2, then N^2-1, then N^2-2 and so on. Thus making the space complexity N^2N, but we don’t need all the N^2 options each time. Therefore, the overall space complexity is O(N^2).

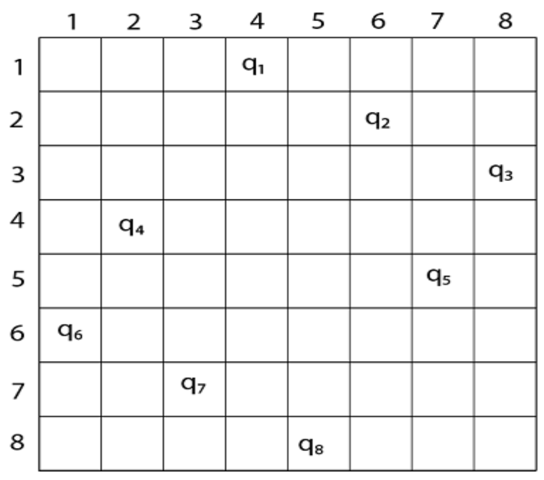


Figure 5.1

```
Algorithm place(k,i)
{
for j:=1 to k-1 do
if ((x[j]=i) or (abs(x[j]-i) = abs(j-k)) then
return false;
return true;
}
Algorithm nqueens(k,n)
{
    for i:=1 to n do
    {
        if (place(k,i) then
        {
            X[k]:=i;
            if (k=n) then
            write(x[i:n]);
            else
            nqueens(k+1,n);
        }
    }
}
```

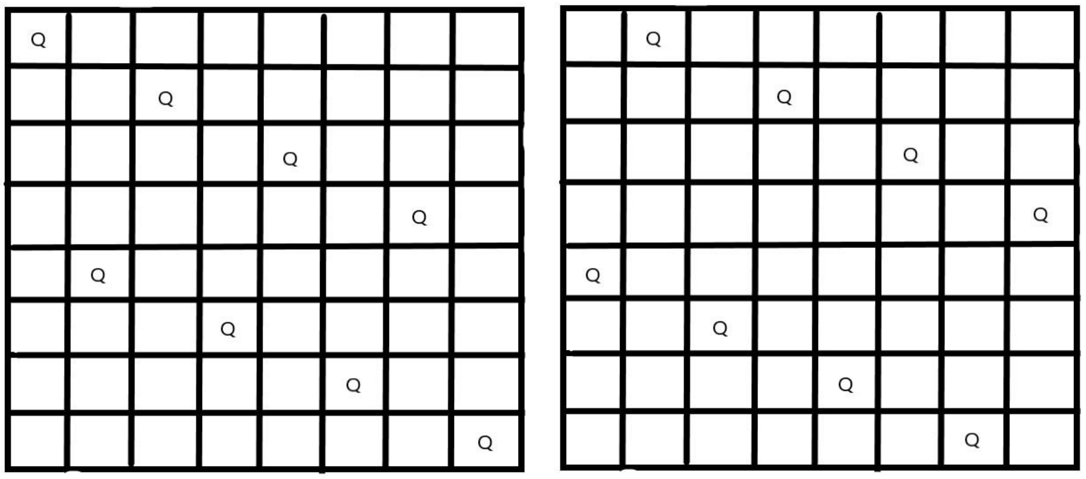


Figure 5.2
These are two possible solutions from the entire solution set for the 8 queen problem.

5.3 Hamiltonian Circuit or Cycle

The term Hamiltonian comes from William Hamiltonian, who invented (a not very successful) board game he termed the "icosian game", which was about finding Hamiltonian cycles on the dodecahedron graph (and possibly its subgraphs).

Hamiltonian Cycle or Circuit in a graph **G** is a cycle that visits every vertex of **G** exactly once and returns to the starting vertex.

- If graph contains a Hamiltonian cycle, it is called **Hamiltonian graph** otherwise it is **non-Hamiltonian**.
- Finding a Hamiltonian Cycle in a graph is a well-known NP-complete problem, which means that there's no known efficient algorithm to solve it for all types of graphs. However, it can be solved for small or specific types of graphs.
- The Hamiltonian Cycle problem has practical applications in various fields, such as **logistics, network design, and computer science**.

5.3.1 Hamiltonian Path

Hamiltonian Path in a graph **G** is a path that visits every vertex of G exactly once and **Hamiltonian Path** doesn't have to return to the starting vertex. It's an open path.

- Similar to the **Hamiltonian Cycle** problem, finding a **Hamiltonian Path** in a general graph is also NP-complete and can be challenging. However, it is often an easier problem than finding a Hamiltonian Cycle.
- Hamiltonian Paths have applications in various fields, such as **finding optimal routes in transportation networks, circuit design, and graph theory research**.

Problems Statement: Given an undirected graph, the task is to determine whether the graph contains a Hamiltonian Circuit/Cycle or not. If it contains, then prints the path.

Example:

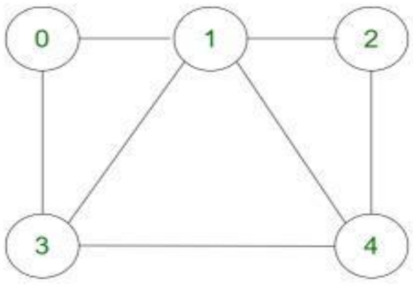


Figure 5.3

Output: {0, 1, 2, 4, 3, 0}.

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations. So the overall Time Complexity of this approach will be **O(N!).**

5.3.2 Hamiltonian Circuit / Cycle using Backtracking Algorithm:

Create an empty path array and add vertex **0** to it. Add other vertices, starting from the vertex **1**. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return **false**.

Let's find out the Hamiltonian cycle for the following graph:

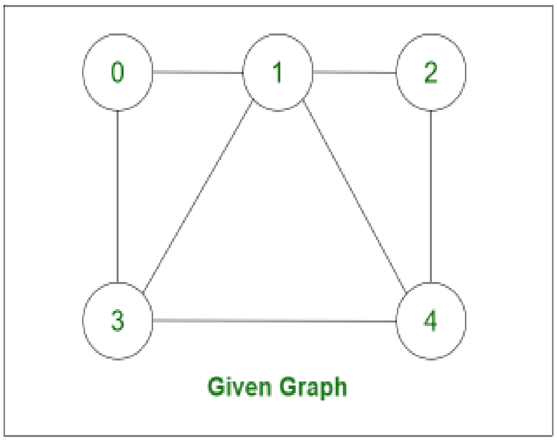


Figure 5.4

- Start with the node 0.
- Apply DFS for finding the Hamiltonian path.
- When base case reach (i.e. **total no of node traversed == V (total vertex)**):
    - Check weather current node is a neighbour of starting node.
    - As node **2** and node **0** are not neighbours of each other so return from it.

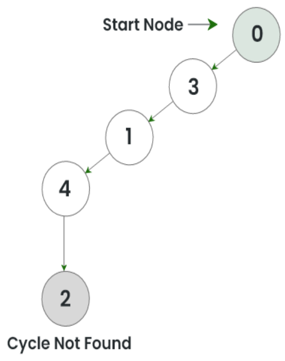


Figure 5.5

Starting from start node 0 calling DFS

- As cycle is not found in path {0, 3, 1, 4, 2}. So, return from node 2, node 4.

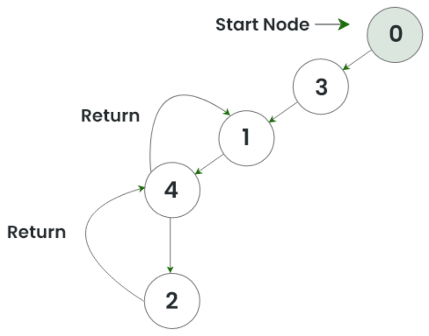


Figure 5.6

- Now, explore another option for node 1 (i.e node 2)
- When it hits the base condition again check for Hamiltonian cycle
- As node 4 is not the neighbour of node 0, again cycle is not found then return.

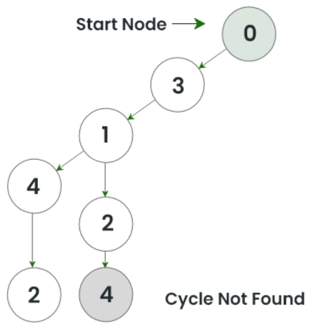


Figure 5.7

Return from node 4, node 2, node 1.

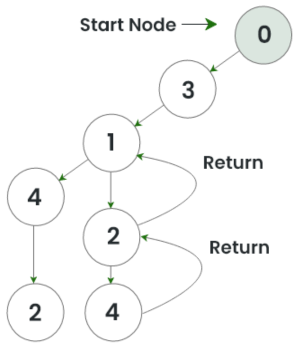


Figure 5.8

Now, explore other options for node 3.

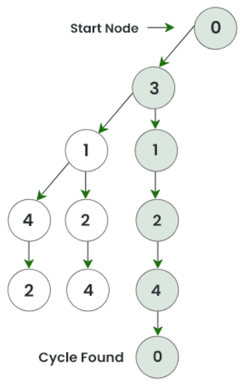


Figure 5.9

Found the Hamiltonian Cycle

- In the Hamiltonian path **{0,3,4,2,1,0}** we get cycle as node 1 is the neighbour of node 0.
- So print this cyclic path.
- This is our Hamiltonian Circuit / Cycle.

5.4 Branch and Bound - Introduction:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node.

Branch and Bound differs from backtracking in two important points:

It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node.

- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

General Methods in branch and Bound

- FIFO branch and bound search

- LIFO branch and bound search
- LC (Least Cost) branch and bound search / LCBB search

5.4.1 FIFO branch and bound search:

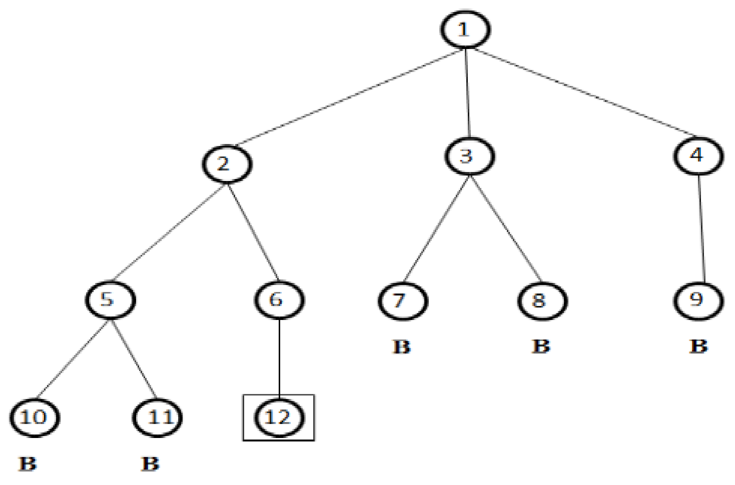


Figure 5.10 State space tree

Assume that the node 12 is an answer node (solution).

In FIFO search first we start with node 1. i.e. E-Node is 1.
Next we generate the children of node 1. We place all the live nodes in queue

2	3	4		

Now we delete an element from queue i.e. node 2, next and node 2 becomes E-node and generates its children and places those live nodes in queue.

3	4	5	6	

Next, delete an element from queue and considering it as E-node. i.e. E-node =3 and its children are generated. 7, 8 are children of 3 and these live nodes are killed by bounding functions. So we will not include in queue.

4	5	6		

Again delete an element from Q. considering it as E-node, generate the children of node 4. Node 9 is generated and killed by bounding function.

5	6			

Again delete an element from Q. generate children of node 5, i.e. nodes 10 and 11 are generated and killed by bounding function, last node in the queue is 6.

6				

The child of node 6 is 12 and it satisfies the conditions of the problem which is the answer node, so search terminates.

5.4.2 LIFO branch and bound search:

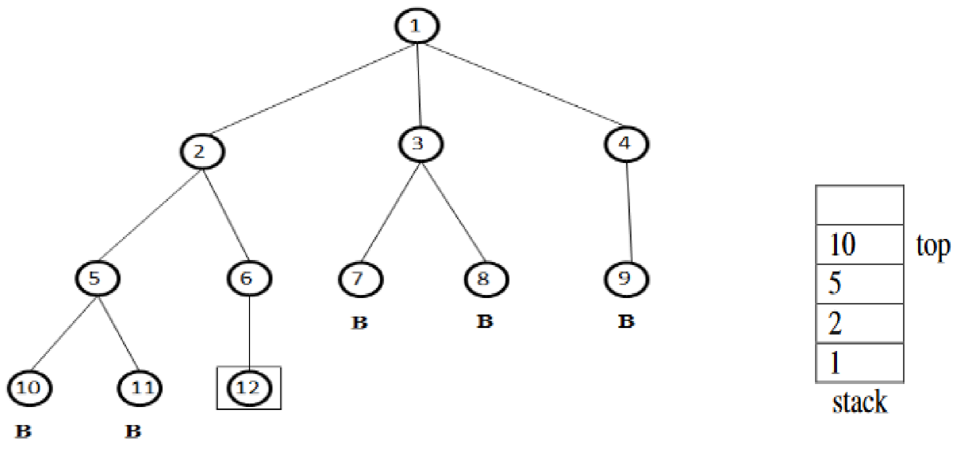


Figure 5.11 State space tree
For LIFO branch and bound we use the data structure stack. Initially stack is empty.
The order is 1,2,5,10,11,6,12

5.4.3 LC (Least Cost) branch and bound search / LCBB search:

In this we use a ranking function or cost function, which is denoted by $\hat{c}(x)$. We generate the children of the E-node, among these live nodes, we select a node which has minimum cost. By using the ranking function we will calculate the cost of each node.
Note that the ranking function $\hat{c}(x)$ or cost function which depends on the problem.
Initially we take the node 1 as E-node.

Generate children of node 1, the children are 2, 3, and 4. By using the ranking function we calculate the cost of 2, 3 and 4 nodes as $\hat{c}(2) = 2$, $\hat{c}(3) = 3$, $\hat{c}(4) = 4$.Now we select a node which has minimum cost i.e. node 2. For node 2 the children are 5 and 6. $\hat{c}(5) = 4$, $\hat{c}(6) = 1$. Now we select node 6 as cost is less. Generate children of 6 i.e. 12 and 13. We select node 12 since its cost is 1 i.e. minimum. More over node 12 is the answer node. So we terminate search process.

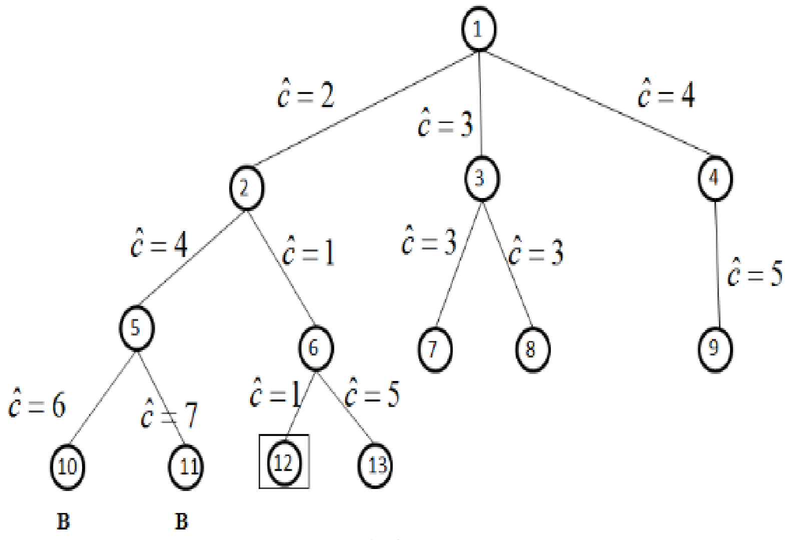


Figure 5.12 State space tree

5.5 Assignment Problem

Assigning n people to n jobs so that the total cost of the assignment is as small as possible. Select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

$$C = \begin{array}{cccc|l} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ 9 & 2 & 7 & 8 & \text{person } a \\ 6 & 4 & 3 & 7 & \text{person } b \\ 5 & 8 & 1 & 8 & \text{person } c \\ 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

Figure 5.13

Lb= the sum of the smallest elements in each of the matrix’s rows.

For the instance here, this sum is 2 + 3 + 1 + 4 = 10.

The lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a.

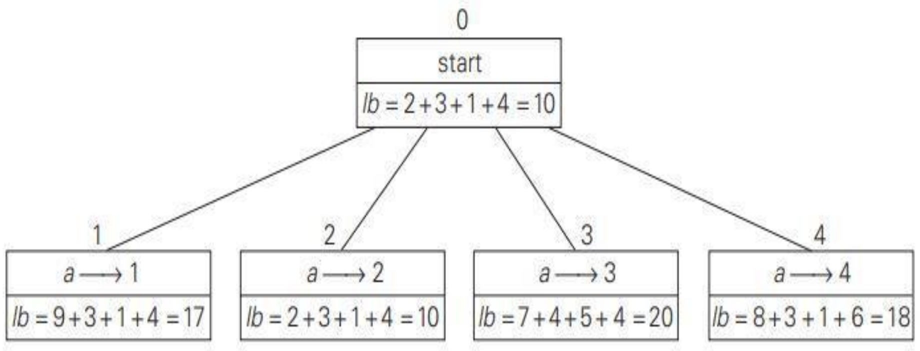


Figure 5.14

The most promising of them is node 2 because it has the smallest lower bound value. We branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column the three different jobs that can be assigned to person b.

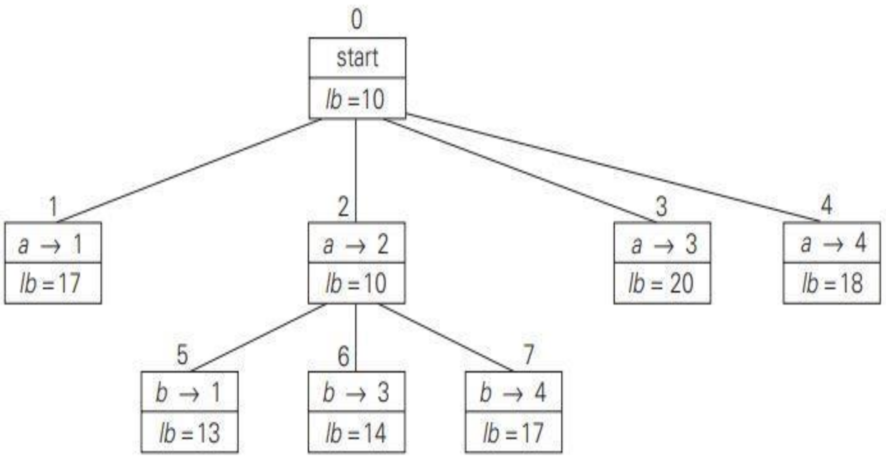


Figure 5.15

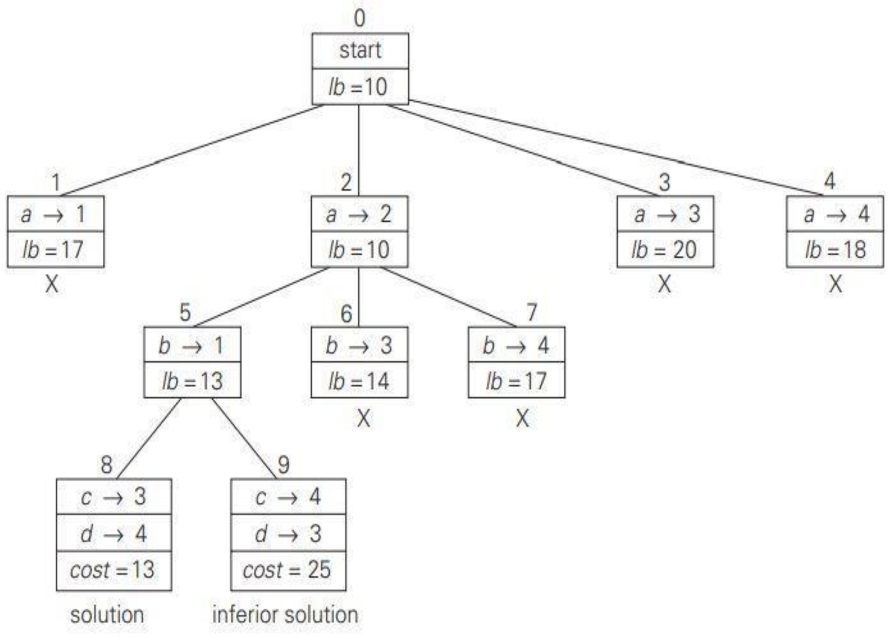


Figure 5.16

5.6 Traveling Salesman Problem:

TSP includes as a salesperson who has to visit a number of cities during a tour and the condition is to visit all the cities exactly once and return back to the same city where the person started.

Basic steps:

Let G=(V,E) be a direct graph defining an instance of the TSP.

1. This graph is first represented by a cost matrix where

cij = the cost of edge , if there is a path between vertex i and vertex j

cij = ∞ , if there is no path

2. Convert cost matrix into reduced matrix i.e.,every row and column should contain at least one zero entry.

3. Cost of the reduced matrix is the sum of elements that are subtracted from rows and columns of cost matrix to make it reduced.

4. Make the state space tree for reduced matrix.

5. To find the next E-node, find the least cost valued node by calculating the reduced cost matrix with every node.

6. If (i, j) edge is to be included, then there are three conditions to accomplish this task:

1. Change all entries in row i and column j of A to ∞.

2. set A [j, 1] =∞

3. Reduce all rows and columns in resulting matrix except for rows and columns containing ∞.

7. Calculate the cost of the matrix where

cost = L + cost (i, j) + r

where L = cost of original reduced cost matrix and

r= new reduced cost matrix

8. Repeat the above steps for all the nodes until all the nodes are generated and we get a path.

If there are n cities and cost of travelling from one city to another city is given. A sales person has to start from any one of the city and has to visit all the cities exactly once and has to return to the starting place with shortest distance or minimum cost.

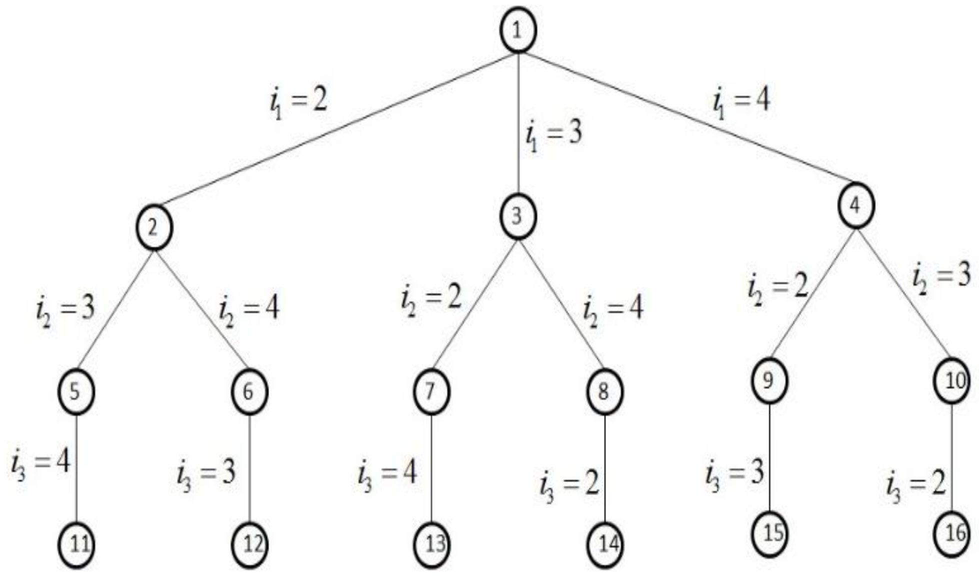


Figure 5.17 State space Tree for the travelling salesperson problem with n=4 and i0 = i4 = 1 i.e. the graph contains 4 nodes and start and end nodes are node 1.

Initially the sales person starts from node 1. Which is E-Node.

As there are 4 nodes the sales person starts at node 1 and reaches node 1 through nodes 2, 3 and 4.

From node1 there is a possibility of travelling to node 2 or node 3 or node 4. So it generates 3 live nodes namely node 2, node 3 and node 4.

Now one of nodes 2 or 3 or 4 becomes E-node depending on the cost. For each node a cost value will be calculated. Depending on the cost the next E-node is selected. The process is continued.

Let G=(V,E) be a directed graph defining an instance of the travelling sales person problem.

Let Cij be the cost of the edge (i,j) and cij= ∞ if (i,j)∉E(G) and let |V |=n

Assume that every tour starts and ends at vertex 1.

To use least cost branch and bound to search the travelling sales person state space tree, we must define a cost function c(x) and two other functions $\hat{c}(x)$ and $\hat{u}(x)$ such that $\hat{c}(x) <= c(x) <= \hat{u}(x)$

5.6.1 LCBB Travelling Salesperson problem:

The cost matrix of a graph consisting of 5 vertices is given under. Each entry in the cost matrix is considered as the cost to travel from one node to another node. We need to find the least cost path from node 1 to node 1 through all other nodes such that the remaining nodes will be visited only once using Least Cost Branch and Bound Method.

Cost matrix:

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 20 & 30 & 10 & 11 \\ 2 & 15 & \infty & 16 & 4 & 2 \\ 3 & 3 & 5 & \infty & 2 & 4 \\ 4 & 19 & 6 & 18 & \infty & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty \end{array}$$

Figure 5.18

Solution

A row is said to be reduced iff it contains at least one 0 and all remaining entries are non-negative.

A column is said to be reduced iff it contains at least one 0 and all remaining entries are non-negative.

A matrix is reduced iff every row and column is reduced.

Row Reduction :

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix} \quad \Rightarrow \quad \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 2 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Figure 5.19

Column Reduction:

$$\begin{array}{c}\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 2 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \\ \begin{matrix} 1 & - & 3 & - & - \end{matrix}\end{array} \Rightarrow \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 2 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Figure 5.20

Total amount subtracted = r + c = 21 + 4 = 25

ĉ(1)=25

25 will be the minimum cost to travel from node 1 as source and destination and nodes 2,3,4,5 as intermediate nodes. So node 1 is root node and it is E-node. As there is a possibility of selecting path in either of nodes 2 or 3 or 4 or 5 it generates node 2, node 3, node 4 and node 5 as live nodes.

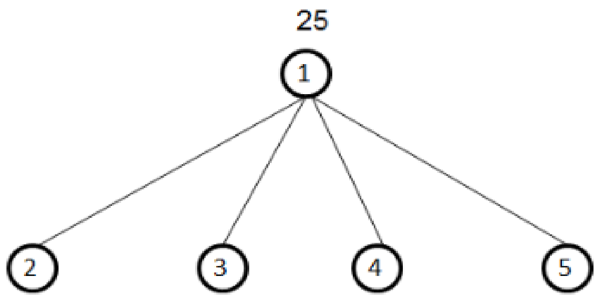


Figure 5.21 Part of a state space tree

Now find minimum cost of nodes 2,3,4 and 5.i.e. ĉ (2), ĉ (3), ĉ (4) and ĉ (5)

Consider the path (1,2) node 2: The reduced cost matrix may be obtained as follows.

1) Change all entries in row i and column j of A to ∞ . This prevents any more edges leaving vertex i or entering vertex j.

2) Set A(j,1) to ∞ . This prevents the use of edge <j,1>.

3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

Change all entries of first row and second column of reduced matrix to ∞.

Assign A[2,1]= ∞ .

Reduce row and reduce column

We get the following reduced cost matrix:

$$A = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 2 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Figure 5.22

The reduced cost matrix after applying row reduction and column reduction i.e. ⇒r =0

ĉ(2)= ĉ(1) + A(1,2) + r = 25 + 10 + 0 = 35

Consider the path (1,3) node 3: Change all entries of first row and third column of reduced matrix to ∞ and set A(3,1) to ∞ .

$$A = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 2 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Figure 5.23

Applying row reduction and column reduction. ⇒ r = 11 +0=11

ĉ(3)= ĉ(1) + A(1,3) + r = 25 + 17 + 11 = 53

Consider the path (1,4) node 4 : Changing all entries of first row and fourth column to ∞ and set A(4,1) to ∞ .

$$A = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 2 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Figure 5.24

Applying row reduction and column reduction, ⇒r =0

ĉ(4)= ĉ(1) + A(1,4) + r = 25 + 0 + 0 = 25

Consider the path (1,5) node 5: Changing all entries of first row and fifth column to ∞ and set A(5,1) to ∞ . Reduce row and columns.

$$A=\begin{bmatrix}\infty & 10 & 17 & 0 & 1\\ 12 & \infty & 11 & 2 & 0\\ 0 & 2 & \infty & 0 & 2\\ 15 & 3 & 12 & \infty & 0\\ 11 & 0 & 0 & 12 & \infty\end{bmatrix}\Rightarrow \begin{array}{c}\left[\begin{array}{ccccc}\infty & \infty & \infty & \infty & \infty\\ 12 & \infty & 11 & 2 & \infty\\ 0 & 3 & \infty & 0 & \infty\\ 15 & 3 & 12 & \infty & \infty\\ \infty & 0 & 0 & 12 & \infty\end{array}\right]\begin{array}{c} \\ 2\\ \\ 3\\ \\ \end{array}\\ \hfill 5\end{array} \Rightarrow \begin{bmatrix}\infty & \infty & \infty & \infty & \infty\\ 10 & \infty & 9 & 0 & \infty\\ 0 & 3 & \infty & 0 & \infty\\ 12 & 0 & 9 & \infty & \infty\\ \infty & 0 & 0 & 12 & \infty\end{bmatrix}$$

Figure 5.25

Applying row reduction and column reduction, ⇒ r=5

$\hat{c}(5)= \hat{c}(1) + A(1,5) + r = 25 + 1 + 5 = 31$

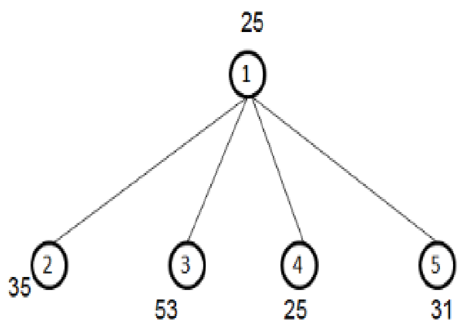


Figure 5.26

ĉ(2)= 35, ĉ(3) = 53, ĉ(4) = 25 and ĉ(5) = 31

Since the minimum cost is 25, select node 4.

The matrix obtained for path (1,4) is considered as reduced cost matrix.

$$A=\begin{bmatrix}\infty & \infty & \infty & \infty & \infty\\ 12 & \infty & 11 & \infty & 0\\ 0 & 3 & \infty & \infty & 2\\ \infty & 3 & 12 & \infty & 0\\ 11 & 0 & 0 & \infty & \infty\end{bmatrix}$$

Figure 5.27

Now node 4 becomes E-node. Its children 6,7, and 8 are generated. The cost values are calculated for nodes 6,7 and 8.

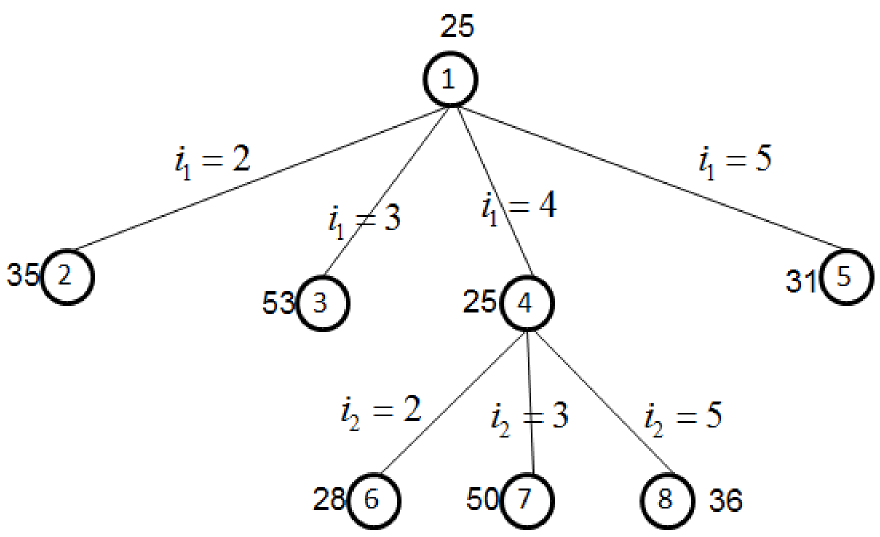


Figure 5.28 Part of a state space tree

Consider the path(1,4,2) node 6: Change all entries of fourth row and second column of reduced matrix A to ∞ and set A(2,1) to ∞.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Figure 5.29

Apply row reduction and column reduction ⇒r =0

ĉ(2)= ĉ(4) +A(4,2) + r = 25+3+0=28

Consider the path(1,4,3) node 7: Change all entries of fourth row and third column of reduced matrix A to ∞ and set A(3,1) to ∞

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix} \Rightarrow \underset{\begin{array}{ccccc} 11 & & & & \end{array}}{\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}} \begin{array}{c} \\ \\ 2 \\ \\ \\ \end{array} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Figure 5.30

Apply row reduction and column reduction ⇒r=2+11=13

ĉ(3)= ĉ(4) +A(4,3) + r = 25+12+13=50

Consider the path(1,4,5) node 8: Change all entries of fourth row and fifth column of reduced matrix A to ∞ and set A(5,1) to ∞

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \begin{array}{l} \\ 11 \\ \\ \\ \\ \end{array} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Figure 5.31

Apply row reduction and column reduction ⇒r=11+0=11

ĉ(5)= ĉ(4) +A(4,5) + r = 25+0+11=36

Since the minimum cost is 28, select node 2.

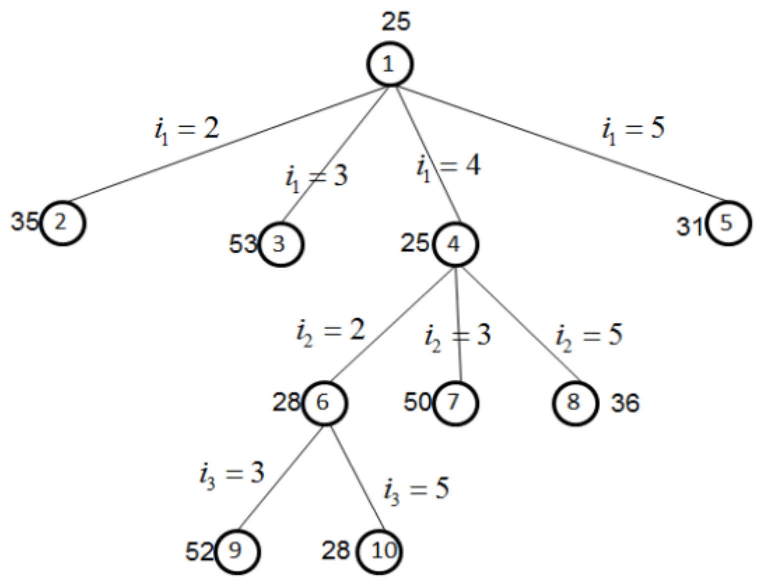


Figure 5.32 Part of the state space tree

Consider the path(1,4,2,3) node 9: Change all entries of second row and third column to ∞ and set A(3,1) to ∞

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{array}{c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \begin{array}{l} \\ \\ 2 \\ \\ \\ \end{array} \\ \begin{array}{lllll} 11 & & & & \end{array} \end{array} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Figure 5.33

Apply row reduction and column reduction ⇒r=2+11=13

ĉ(3)= ĉ(2) +A(2,3) + r = 28+11+13=52

Consider the path (1,4,2,5) node 10: Change all entries of second row and fifth column to ∞ and set A(5,1) to ∞

$$A=\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Figure 5.34

Apply row reduction and column reduction ⇒r=0

ĉ(3)= ĉ(5) +A(5,3) + r = 28+0+0=28

Since minimum cost is 28, select node 5.

The matrix obtained for path (2,5) is considered as reduced cost matrix.

Consider the path (1,4,2,5,3) node 11: Change all entries of fifth row and third column to ∞ and set A(3,1) to ∞

$$A=\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Figure 5.35

Apply row reduction and column reduction ⇒ r=0

ĉ(3)= ĉ(5) +A(5,3) + r = 28+0+0=28

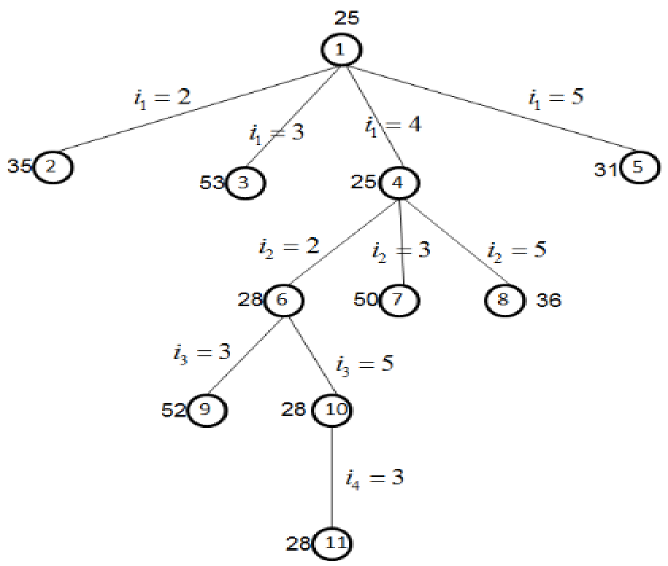


Figure 5.36 Final State Space Tree

The path is 1-4-2-5-3-1

Minimum cost=10+6+2+7+3=28

LCBB terminates with 1,4,2,5,3,1 as the shortest length tour.

5.7 0/1 Knapsack problem::LCBB solution:

The 0/1 knapsack problem states that, there are n objects given and capacity of knapsack is m. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned

The 0/1 knapsack problem can be stated as

$Max\ z = p_1 x_1 + p_2 x_2 + \ldots.\ p_n x_n$

$w_1 x_1 + w_2 x_2 + \ldots.. + w_n x_n \leq m$

X_i=0 or 1.

A branch and bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch and bound deals with only minimization problems. We modify the knapsack problem to the minimization problem.

The modified problem is

$Min\ z = p_1 x_1 + p_2 x_2 + \ldots.\ p_n x_n$

$w_1 x_1 + w_2 x_2 + \ldots.. + w_n x_n \leq m$

X_i=0 or 1.

Let $\hat{c}(x)$ and $\hat{u}(x)$ are the two cost functions such that $\hat{c}(x)$<= c(x)<= $\hat{u}(x)$ satisfying the requirements where

$$c(x) = -\sum_{i=1}^{n} pixi$$

The c(x) is the cost function for answer node x, which lies between two functions called lower and upper bounds for the cost function c(x). The search begins at the root node. Initially we compute the lower and upper bound at root node called $\hat{c}(1)$ and $\hat{u}(1)$. Consider the first variable x1 to take a decision. The x1 takes values either 0 or 1. Compute the lower and upper bounds in each case of variable. These are the nodes at the first level. Select the node whose cost is minimum i.e.

c(x)=min{*c(lchild(x), c(rchild(x))*}

c(x)=min{ $\hat{c}(2)$,$\hat{c}(3)$}

The problem can be solved by making a sequence of decisions on the variables $x_1,x_2,\ldots,x_n$ level wise. A decision on the variable x_i involves determining which of the values 0 or 1 is to be assigned, to it by defining c(x) recursively.

The path from root to the leaf node whose height is maximum is selected and is the solution space for the knapsack problem**.**

5.7.1 Problem1) LCBB.

Consider the knapsack instance n=4, (p1,p2,p3,p4)=(10,10,12,18),

(W1,W2,W3,W4)=(2,4,6,9), m=15. Let us trace the working of an LCBB search.

Solution: Let us use the fixed tuple formulation.

The search begins with root node as E-node.

cp - current profit

cw- current weight

k – k number of decisions

m - Capacity of knapsack

Algorithm Ubound(cp,cw,k,m)

```
{
        b := cp;
        c := cw;
        for i:= k+1 to n do
        {
                if ( c + w[i] <= m ) then
                {
                c := c + w[i];
                b := b - p[i];
                }
        }
        return b;
}
```

Function U(.) for Knapsack problem

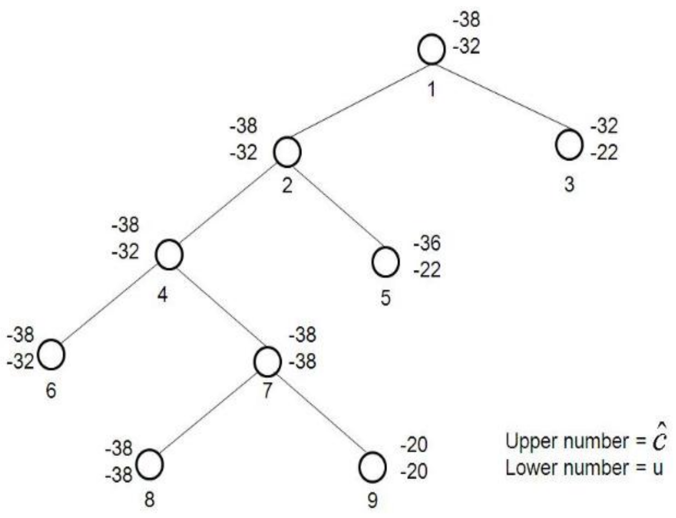


Figure 5.37 LC Branch and Bound tree

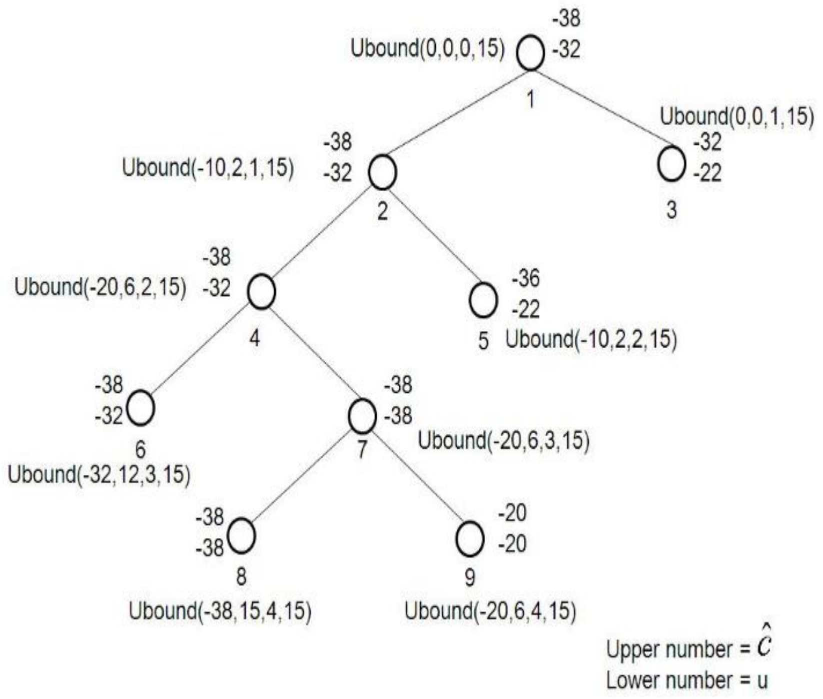


Figure 5.38 shows the Upper bound function call for each node

Node 1	Ubound(0,0,0,15)			
	i=1	i=2	i=3	i=4
	c=2	c=6	c=12	
	b=-10	b=-20	b=-32	
	U(1)=-32			
	$\hat{c}$(1)= -32+3/9 x 18= -38			
	To calculate lower bound we allow fractions			

Node 2	Ubound(-10,2,1,15)		x1=4
	i=2	i=3	i=4
	c=6	c=12	
	b=-20	b=-32	

U(1)=-32
$\hat{c}(2)$= -32+3/9 x 18= -38
To calculate lower bound we allow fractions

Node 3 Ubound(0,0,0,15)

	i=2	i=3	i=4
	c=4	c=10	
	b=-10	b=-22	

U(3)=-22
$\hat{c}(3)$= -22+5/9 x 18= -32
To calculate lower bound we allow fractions

Node 4 Ubound(-20,6,2,15)

	i=3	i=4
	c=12	
	b=-32	

U(4)=-32
$\hat{c}(4)$= -32+3/9 x 18= -38
To calculate lower bound we allow fractions

Node 5 Ubound(-10,2,2,15)

	i=3	i=4
	c=8	
	b=-22	

U(5)=-22
$\hat{c}(5)$= -22+7/9 x 18= -36
To calculate lower bound we allow fractions

Node 6 Ubound(-32,12,3,15)
i=4

U(6)=-32
$\hat{c}(6)$= -32+3/9 x 18= -38
To calculate lower bound we allow fractions

Node 7 Ubound(-20,6,3,15)
i=4
c=15
b=-38
U(7)=-38
$\hat{c}(7)$= -38

Node 8 Ubound(-38,15,4,15)

U(8)=-38
$\hat{c}(8)$= -38

Node 8 Ubound(-20,6,4,15)

U(1)=-20

$\hat{c}(1) = -20$

Node 8 is a solution node (solution vector)

X1=1

X2=1

X3=0

X4=1

p1x1+p2x2+p3x3+p4x4

10x1+10=1+12x0+18x1

=38

We need to consider all n items.

The tuple size is n

5.7.2 0/1 Knapsack Problem

Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and (w1, w2, w3, w4) = (2, 4, 6, 9).

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node. Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.

Profit = P1 + P2 + P3 = 10 + 10 + 12

So, Upper bound = 32

To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = 10 + 10 + 12 + (3/9 X 18) = 32 + 6 = 38

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, Upper bound (U) = -32

Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.

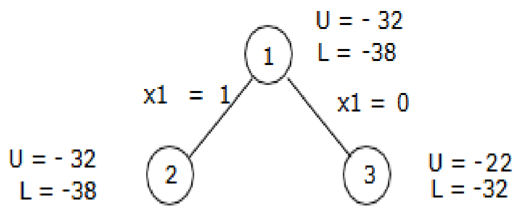


Figure 5.39
Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, x1= 1, means we should place first item in the knapsack.
U = 10 + 10 + 12 = 32, make it as -32
L = 10 + 10 + 12 + (3/9) x 18 = 32 + 6 = 38, make it as -3

For node 3, x1 = 0, means we should not place first item in the knapsack.
U = 10 + 12 = 22, make it as -22
L = 10 + 12 +(5/9) x 18 = 10 + 12 + 10 = 32, make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3
For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

Choose node 2, since it has minimum difference value of 6.

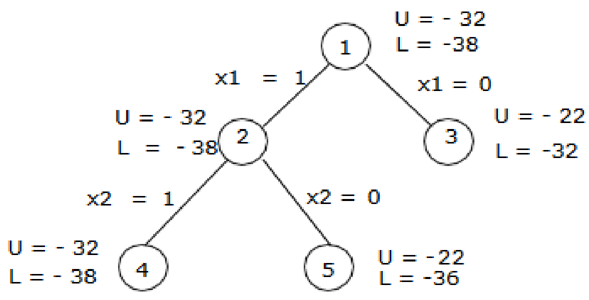


Figure 5.40
Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5

For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14

Choose node 4, since it has minimum difference value of 6.

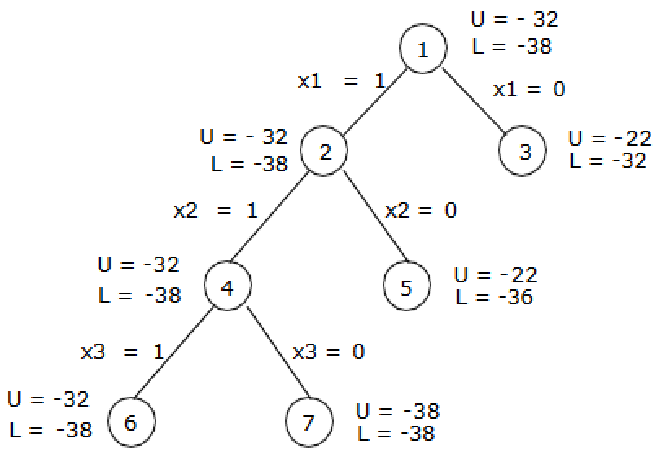


Figure 5.41

Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9

For node 6, U – L = -32 + 38 = 6
For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.

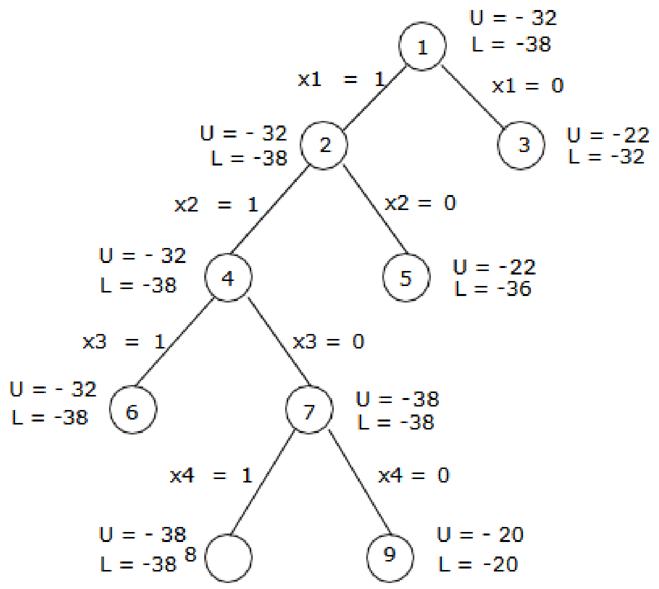


Figure 5.42

Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 = 0
For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1 -> 2 -> 4 -> 7 -> 8

X1 = 1
X2 = 1
X3 = 0
X4 = 1

The solution for 0/1 Knapsack problem is (x1, x2, x3, x4) = (1, 1, 0, 1)
Maximum profit is:

$$\begin{aligned} \Sigma P_i\, x_i &= 10 \text{ x } 1 + 10 \text{ x } 1 + 12 \text{ x } 0 + 18 \text{ x } 1 \\ &= 10 + 10 + 18 = 38. \end{aligned}$$

5.8 NP-complete Problems:

NP-complete problems are a subset of the larger class of **NP (nondeterministic polynomial time) problems**. **NP** problems are a class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic Machine. A problem **L** in **NP** is **NP-complete** if all other problems in **NP** can be reduced to **L** in polynomial time. If any **NP-complete** problem can be solved in polynomial time, then every problem in **NP** can be solved in polynomial time. **NP-complete** problems are the hardest problems in the **NP** set.

A decision problem **L** is **NP-complete** if it follow the below two properties:

1. **L** is in **NP** (Any solution to NP-complete problems can be checked quickly, but no efficient solution is known).

2. Every problem in **NP** is reducible to **L** in polynomial time.

A problem is **NP-Hard** if it obeys Property 2 above and need not obey Property 1. Therefore, a problem is **NP-complete** if it is both **NP** and **NP-hard**.

NP-complete problem, any of a class of computational problems for which no efficient solution algorithm has been found. Many significant computer-science problems belong to this class—e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.

So-called easy, or tractable, problems can be solved by computer algorithms that run in polynomial time; i.e., for a problem of size *n*, the time or number of steps needed to find the solution is a polynomial function of *n*. Algorithms for solving hard, or intractable, problems, on the other hand, require times that are exponential functions of the problem size *n*. Polynomial-time algorithms are considered to be efficient, while exponential-time algorithms are considered inefficient, because the execution times of the latter grow much more rapidly as the problem size increases.

5.9 Clique problem:

Let G be a non-directed graph consisting of a set of vertices V and set of edges E. A maximal complete sub graph of a graph G is a clique. The size of the clique is the number of vertices in it. The max clique problem is an optimization problem that has to determine whether graph has a clique of size atleast k for some integer k.

A **clique** is a subset of vertices of an undirected graph G such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. Cliques are one of the basic concepts of graph theory and are used in many other mathematical problems and constructions on graphs.

The task of finding whether there is a clique of a given size in a graph (the clique problem) is NP complete, but despite this hardness result, many algorithms for finding cliques have been studied.

A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique.

A **maximum clique** of a graph, G, is a clique, such that there is no clique with more vertices. Moreover, the **clique number** ω(G) of a graph G is the number of vertices in a maximum clique in G.

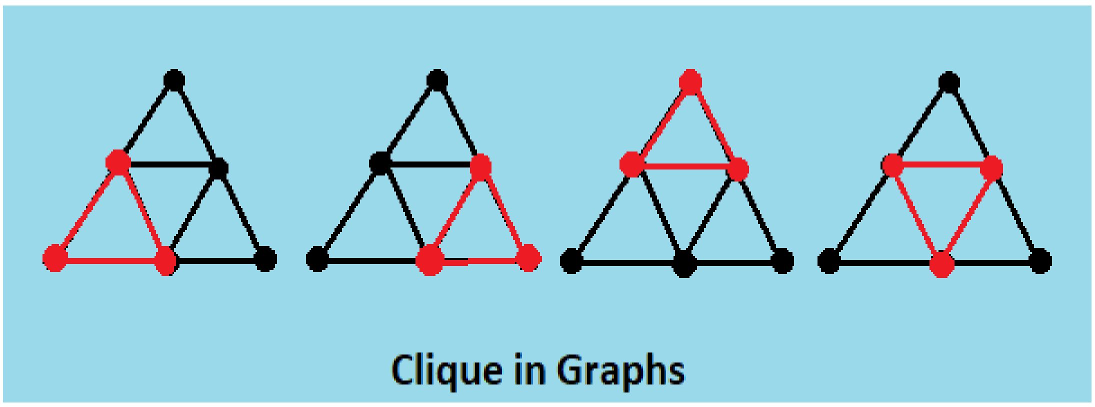


Figure 5.43

The clique problem is the computational problem of finding cliques in a graph. It has several different formulations depending on which cliques, and what information about the cliques,shouldbefound.

Common formulations of the clique problem include finding a maximum clique, finding a maximum weight clique in a weighted graph, listing all maximal cliques, and solving the decision problem of testing whether a graph contains a clique larger than a given size.

5.9.1Finding a single maximal clique

A single maximal clique can be found by a straightforward **greedy algorithm**. Starting with an arbitrary clique (for instance, any single vertex or even the empty set), grow the current clique one vertex at a time by looping through the graph's remaining vertices. For each vertex v that this loop examines, add v to the clique if it is adjacent to every vertex that is already in the clique, and discard v otherwise. This algorithm runs in linear time. Because of the ease of finding maximal cliques, and their potential small size, more attention has been given to the much harder algorithmic problem of finding a maximum or otherwise large clique than has been given to the problem of finding a single maximal clique.

5.9.1.1 Cliques of fixed size

One can test whether a graph G contains a k-vertex clique, and find any such clique that it contains, using a brute force algorithm. This algorithm examines each subgraph with k vertices and checks to see whether it forms a clique. It takes time $O(n^k k^2)$, as expressed using big O notation. This is because there are $O(n^k)$ subgraphs to check, each of which has $O(k^2)$ edges whose presence in G needs to be checked. Thus, the problem may be solved in polynomial time whenever k is a fixed constant. However, when k does not have a fixed value, but instead may vary as part of the input to the problem, the time is exponential.

5.10 Vertex Cover Problem:

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP.

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

Following are some examples.

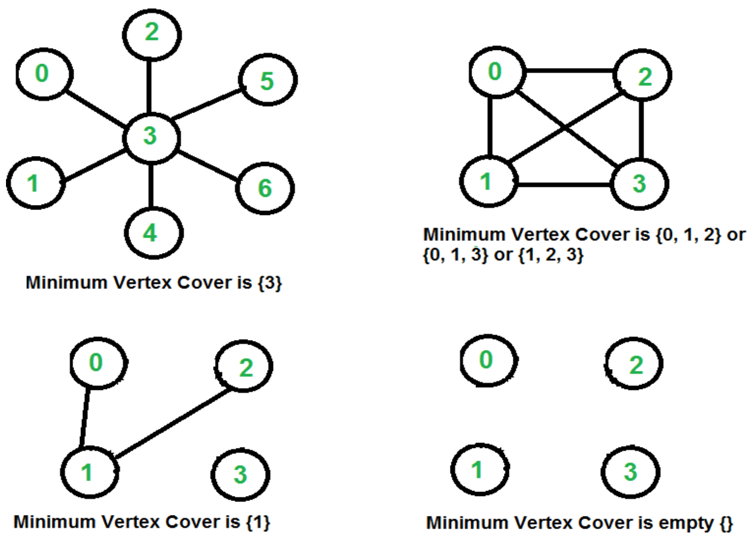

Minimum Vertex Cover is {3}
Minimum Vertex Cover is {0, 1, 2} or {0, 1, 3} or {1, 2, 3}
Minimum Vertex Cover is {1}
Minimum Vertex Cover is empty {}

Figure 5.44

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover***.

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem.

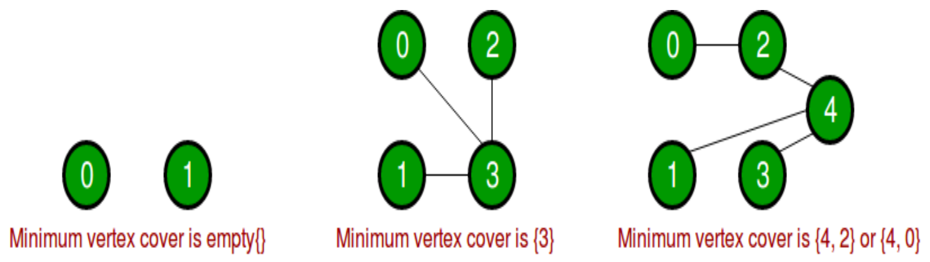

Minimum vertex cover is empty{}
Minimum vertex cover is {3}
Minimum vertex cover is {4, 2} or {4, 0}

Figure 5.45

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are:{0,1,2,{0,1},{0,2},{1,2},{0,1,2}} .

Using each element of this set check whether these vertices cover all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

Approximate Algorithm for Vertex Cover:

1. Initialize the result as {}
2. Consider a set of all edges in given graph. Let the set be E.

3. Do the following while E is not empty.
    a) Pick an arbitrary edge (u,v) from set E and add ‘u’ and ‘v’ to result.
    b) Remove all edges from E which are either incident on u or v.
4. Return result.

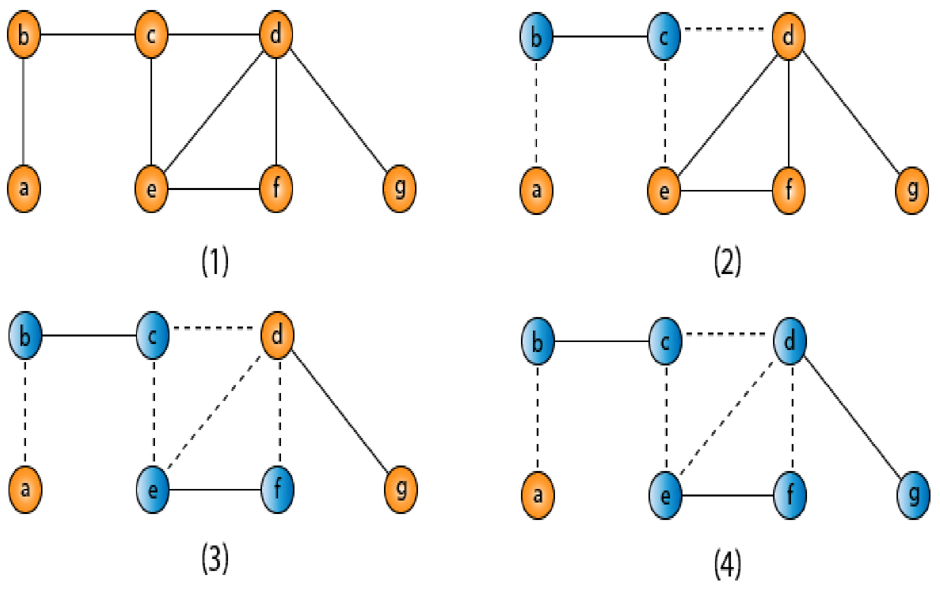


Figure 5.46

VC = {b, c, d, e, f, g}

SAMPLE QUESTIONS:

1. Explain the general method of Branch and Bound.
2. Explain the properties of LC-search
3. Explain the method or reduction to solve TSP problem using Branch and Bound
4. Solve the following instance of travelling sales person problem using LCBB.

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

5. Draw the portion of the state space tree generated by LCBB for the knapsack instance n=5, (p1,p2,p3,p4,p5)=(10,15,6,8,4), (w1,w2,w3,w4,w5)= 4,6,3,4,2) and m=12 Apply branch and bound to the above 0/1 knapsack problem and explain.
6. Explain the principles of FIFO branch and Bound.
7. Draw the portion of the state space tree generated by FIFO branch and bound for the following instances. N=4, m=15, (p1,p2,p3,p4)=(10,10,12,18), (w1,w2,w3,w4)=(12,4,6,9).