A black and white logo

Description automatically generated

**Vidyavardhini’s**

**College of Engineering & Technology**

Vasai Road (W)

**Department of Artificial Intelligence & Data Science Engineering**

**Laboratory Manual**

**Student Copy**

|  |  |  |  |
| --- | --- | --- | --- |
| Semester | IV | **Class** | **S.E** |
| Course Code | CSL401 | | |
| Course Name | Analysis of Algorithms Lab | | |

****

**Vidyavardhini’s College of Engineering & Technology**

**Vision**

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

**Mission**

* To provide technologically inspiring environment for learning.
* To promote creativity, innovation and professional activities.
* To inculcate ethical and moral values.
* To cater personal, professional and societal needs through quality education.

**Department Vision:**

To foster proficient artificial intelligence and data science professionals, making remarkable contributions to industry and society.

**Department Mission:**

* To encourage innovation and creativity with rational thinking for solving the challenges in emerging areas.
* To inculcate standard industrial practices and security norms while dealing with Data.
* To develop sustainable Artificial Intelligence systems for the benefit of various sectors.

**Program Specific Outcomes (PSOs):**

PSO1: Analyze the current trends in the field of Artificial Intelligence & Data Science and convey their finding by presenting / publishing at a national / international forums.

PSO2: Design and develop Artificial Intelligence & Data Science based solutions and applications for the problems in the different domains catering to industry and society.

**Program Outcomes (POs):**

Engineering Graduates will be able to:

* **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
* **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
* **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
* **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
* **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
* **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
* **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
* **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
* **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
* **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
* **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**PO12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Course Objective**

|  |  |
| --- | --- |
| 1 | To introduce the methods of designing and analyzing algorithms |
| 2 | Design and implement efficient algorithms for a specified application |
| 3 | Strengthen the ability to identify and apply the suitable algorithm for the given real-world problem. |
| 4 | Analyze worst-case running time of algorithms and understand fundamental algorithmic problems. |

**Course Outcomes**

|  |  |  |  |
| --- | --- | --- | --- |
| **At the end of the course student will be able to:** | | **Action verbs** | **Bloom’s Level** |
| CSL401.1 | Analyze time complexity of sorting algorithms | Analyze | Apply (Level 3) |
| CSL401.2 | Analyze the complexity of problems solved using divide and conquer approaches | Analyze | Apply  (Level 3) |
| CSL401.3 | Implement greedy algorithms for solving Dijkstras, Minimum spanning tree & fractional knapsack. | Implement | Apply  (Level 3) |
| CSL401.4 | Implement dynamic programming algorithm for All pair shortest path and 0/1 knapsack | Apply | Apply  (Level 3) |
| CSL401.5 | Implement backtracking and branch and bound for 15 puzzle, N queen and sum of subset problem | Apply | Apply  (Level 3) |
| CSL401.6 | Analyze the performance of string-matching techniques | Analyze | Apply  (Level 3) |

**Mapping of Experiments with Course Outcomes**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Sr. No** | **Title** | **CSL 401.1** | **CSL401.2** | **CSL 401.3** | **CSL 401.4** | **CSL 401.5** | **CSL 401.6** |
| 1. | To implement Insertion Sort and Comparative analysis for large values of ‘n’. | 3 | - | - | - | - | - |
| 2. | To implement Selection Sort and Comparative analysis for large values of ‘n’ | 3 | - | - | - | - | - |
| 3. | To implement Quick Sort and Comparative analysis for large values of ‘n’ using DAC technique. | - | 3 | - | - | - | - |
| 4. | To implement Binary Search for ‘n’ number and perform analysis using DAC technique. | - | 3 | - | - | - | - |
| 5. | To implement Fractional Knap Sack using Greedy Method. | - | - | 3 | - | - | - |
| 6. | To implement Prim’s MST Algorithm using Greedy Method. | - | - | 3 | - | - | - |
| 7. | To implement Kruskal’s MST Algorithm using Greedy Method. | - | - | 3 | - | - | - |
| 8. | To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method. | - | - | - | 3 | - | - |
| 9. | To implement Travelling Salesperson Problem using Dynamic Approach. | - | - | - | 3 | - | - |
| 10. | To implement Sub of Subset problem using Backtracking method. | - | - | - | - | 3 | - |
| 11. | To implement 15 puzzle problem using Branch and Bound Method. | - | - | - | - | 3 | - |
| 12. | Implement the Naïve string-matching algorithm and analyse its complexity. | - | - | - | - | - | 3 |

Enter correlation level 1, 2 or 3 as defined below

1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

If there is no correlation put “— “.

**List of Experiments**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr. No** | **Name of Experiments** | **DOP** | **DOC** | **Marks** | **Sign** |
| 1. | To implement Insertion Sort and Comparative analysis for large values of ‘n’. |  |  |  |  |
| 2. | To implement Selection Sort and Comparative analysis for large values of ‘n’ |  |  |  |  |
| 3. | To implement Quick Sort and Comparative analysis for large values of ‘n’ using DAC technique. |  |  |  |  |
| 4. | To implement Binary Search for ‘n’ number and perform analysis using DAC technique. |  |  |  |  |
| 5. | To implement Fractional Knap Sack using Greedy Method. |  |  |  |  |
| 6. | To implement Prim’s MST Algorithm using Greedy Method. |  |  |  |  |
| 7. | To implement Kruskal’s MST Algorithm using Greedy Method. |  |  |  |  |
| 8. | To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method. |  |  |  |  |
| 9. | To implement Travelling Salesperson Problem using Dynamic Approach. |  |  |  |  |
| 10. | To implement Sub of Subset problem using Backtracking method. |  |  |  |  |
| 11. | To implement N queen problem using Branch and Bound Method. |  |  |  |  |
| 12. | Implement the Naïve string-matching algorithm and analyse its complexity. |  |  |  |  |
| **Assignments** | | | | | |
| 1. | Assignment 1: Sorting Algorithms |  |  |  |  |
| 2. | Assignment 2: Divide and Conquer Approach |  |  |  |  |
| 3. | Assignment 3 : Greedy Method Approach |  |  |  |  |
| 4. | Assignment 4 : Dynamic Programming Approach |  |  |  |  |
| 5. | Assignment 5: Backtracking and Branch and Bound |  |  |  |  |
| 6. | Assignment 6: String Matching Algorithms |  |  |  |  |
| **Formative Assessment** | | | | | |
| 1 | TH- Quiz 1: Sorting Algorithms |  |  |  |  |
| 2 | TH- Quiz 2: Divide and Conquer Approach |  |  |  |  |
| 3 | TH- Quiz 3: Greedy Method Approach |  |  |  |  |
| 4 | TH- Quiz 4: Dynamic Programming Approach |  |  |  |  |
| 5 | TH- Quiz 5: Backtracking and Branch and Bound |  |  |  |  |
| 6 | TH - Quiz 6: String Matching Algorithms |  |  |  |  |
| 7 | PR - Quiz 1: Insertion and Selection Sort |  |  |  |  |
| 8 | PR- Quiz 2: Divide and Conquer Approach |  |  |  |  |
| 9 | PR- Quiz 3: Fractional Knapsack ad Job Sequencing, MST |  |  |  |  |
| 10 | PR- Quiz 4: 0\1 Knapsack, TSP, All Pair Shortest Path algorithm |  |  |  |  |
| 11 | PR- Quiz 5: N-Queen and 15 Puzzle Problem |  |  |  |  |
| 12 | TH - Quiz 6: Naïve String , KMP and Rabin Karp |  |  |  |  |

D.O.P: Date of performance

D.O.C : Date of correction

|  |
| --- |
| Experiment No.1 |
| Insertion Sort |
| Date of Performance: |
| Date of Submission: |

**Title**: Insertion Sort

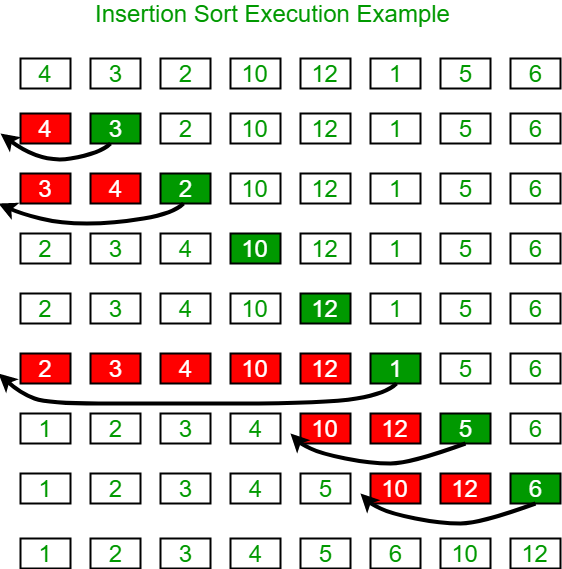
**Aim**: To implement Selection Comparative analysis for large values of ‘n’

**Objective:** To introduce the methods of designing and analysing algorithms

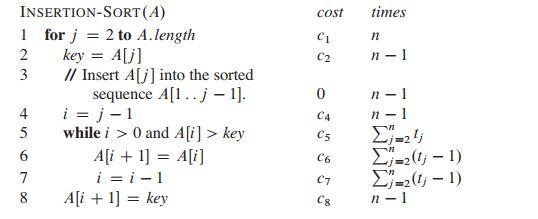
**Theory**:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

**Example:**



**Algorithm and Complexity:**



**Implementation:**

**Conclusion:**

|  |
| --- |
| Experiment No.2 |
| Selection Sort |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 2**

**Title**: Selection Sort

**Aim**: To implement Selection Comparative analysis for large values of ‘n’

**Objective:** To introduce the methods of designing and analyzing algorithms

**Theory**:

Selection sort is a sorting algorithm, specifically an in-place comparison sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sub list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub list of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sub list is empty and the unsorted sub list is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub list, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

**Example**:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4] // and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4] // and place it at beginning of arr[1...4]

11 12 25 22 64

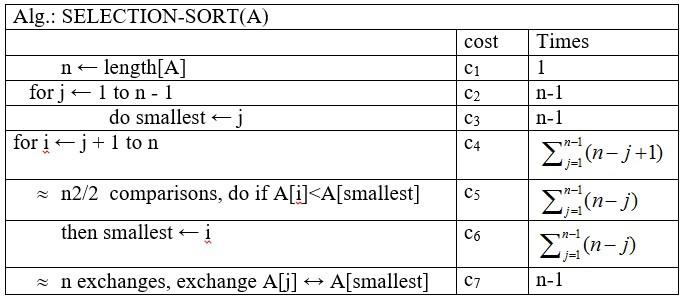
// Find the minimum element in arr[2...4] // and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4] // and place it at beginning of arr[3...4]

11 12 22 **25** 64

**Algorithm and Complexity**:



**Implementation:**

**Conclusion:**

|  |
| --- |
| Experiment No. 3 |
| Quick Sort |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 3**

**Title:** Quick Sort

**Aim:** To implement Quick Sort and Comparative analysis for large values of ‘n’.

**Objective:** To introduce the methods of designing and analyzing algorithms.

**Theory:**

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

1. Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
2. Conquer: Sort the two subsequences recursively using merge sort.
3. Combine: Merge the two sorted subsequence to produce the sorted answer.

Partition-exchange sort or quicksort algorithm was developed in 1960 by Tony Hoare. He developed the algorithm to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.

Quick sort algorithm on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sublists.

1. Elements less than pivot element.

2. Pivot element.

3. Elements greater than pivot element.

Where pivot as middle element of large list. Let’s understand through example:

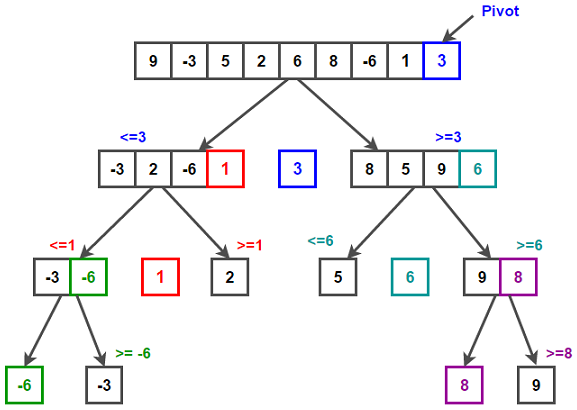
List : 3 7 8 5 2 1 9 5 4

In above list assume 4 is pivot element so rewrite list as:

3 1 2 4 5 8 9 5 7

Here, I want to say that we set the pivot element (4) which has in left side elements are less than and right hand side elements are greater than. Now you think, how’s arrange the less than and greater than elements? Be patient, you get answer soon.

**Example:**



/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element and indicates the

// right position of pivot found so far

for (j = low; j <= high- 1; j++)

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

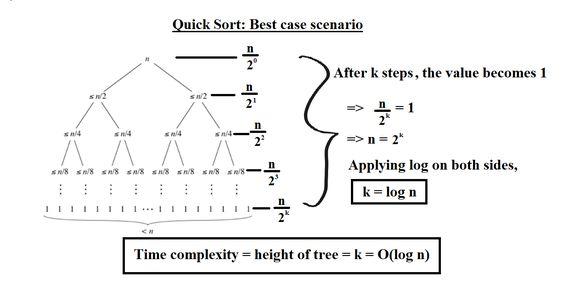
}

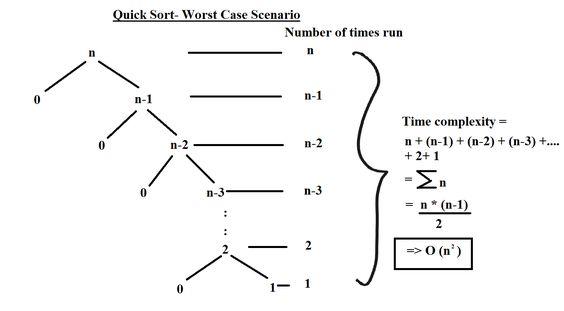
}

swap arr[i + 1] and arr[high])

return (i + 1)

}





**Implementation:**

**Conclusion:** Comment on implementation of Quick sort Algorithm

|  |
| --- |
| Experiment No. 4 |
| Binary Search Algorithm |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 4**

**Title:** Binary Search Algorithm

**Aim:** To study and implement Binary Search Algorithm

**Objective:** To introduce Divide and Conquer based algorithms

**Theory:**

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty

* Binary search is efficient than linear search. For binary search, the array must be sorted, which is not required in case of linear search.
* It is divide and conquer based search technique.
* In each step the algorithms divides the list into two halves and check if the element to be searched is on upper or lower half the array
* If the element is found, algorithm returns.

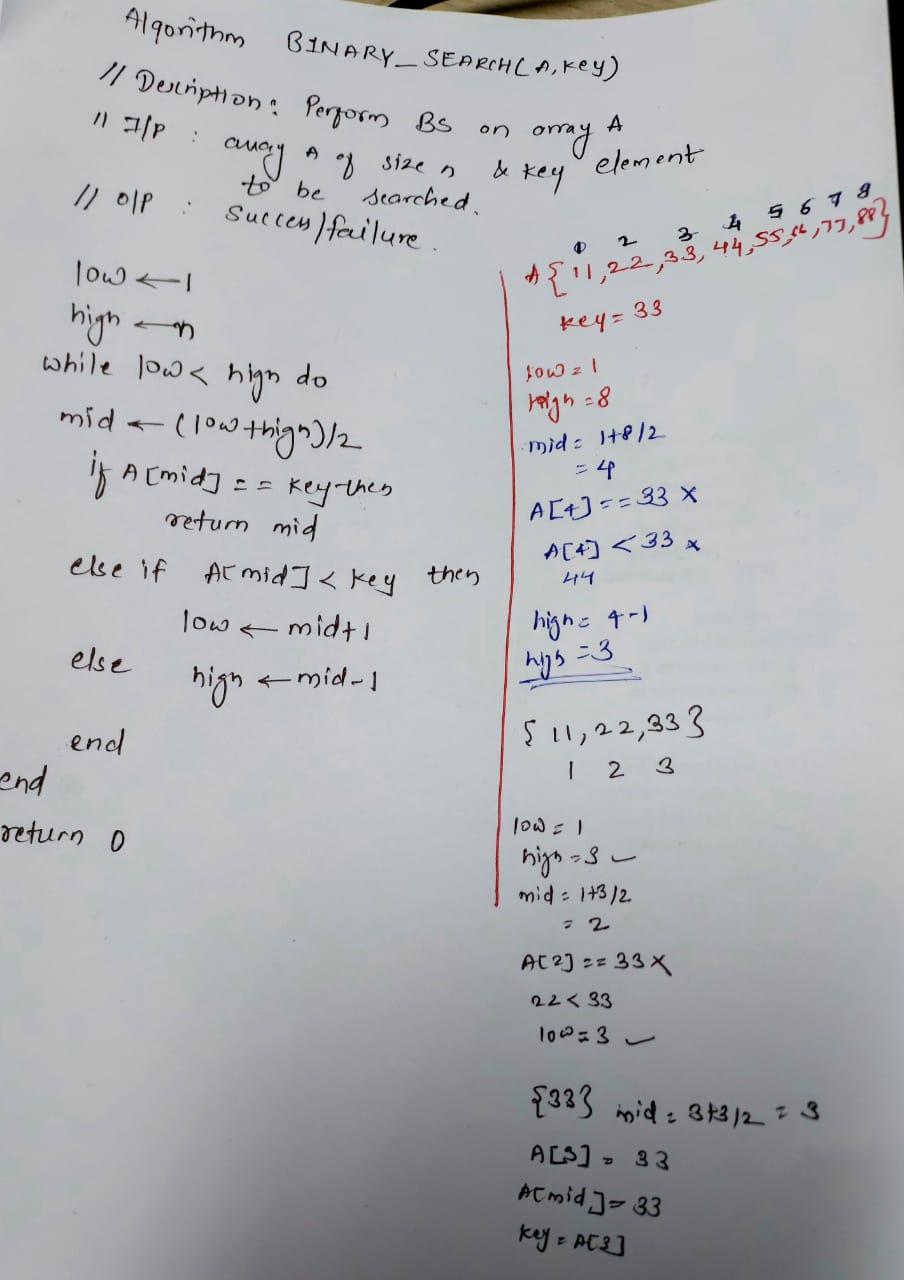


The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

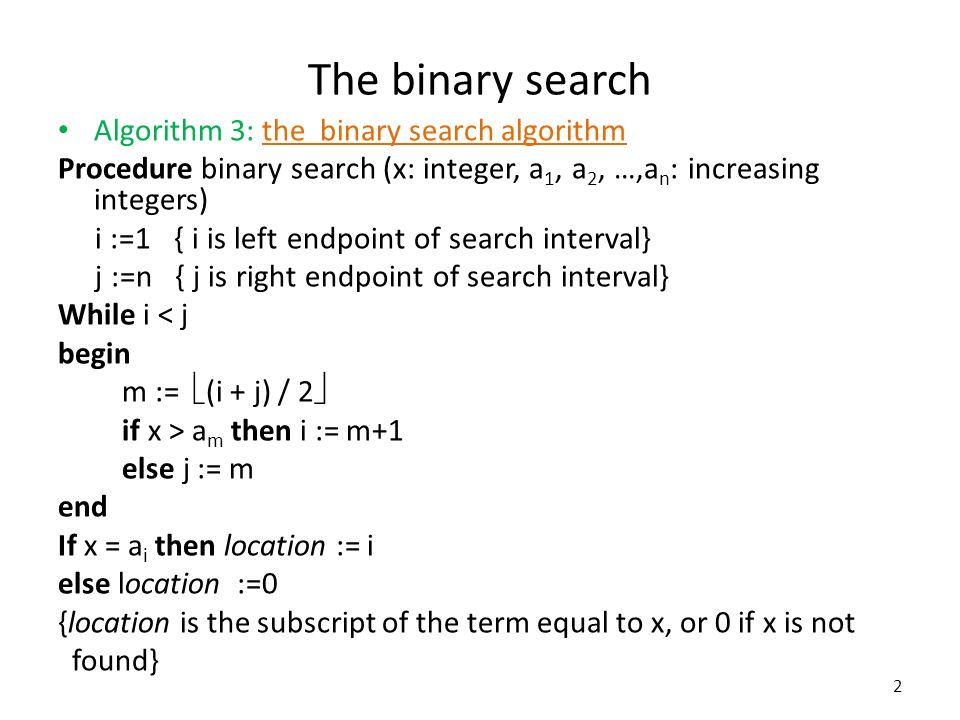
* Compare x with the middle element.
* If x matches with the middle element, we return the mid index.
* Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
* Else (x is smaller) recur for the left half.
* Binary Search reduces search space by half in every iterations. In a linear search, search space was reduced by one only.
* n=elements in the array
* Binary Search would hit the bottom very quickly.

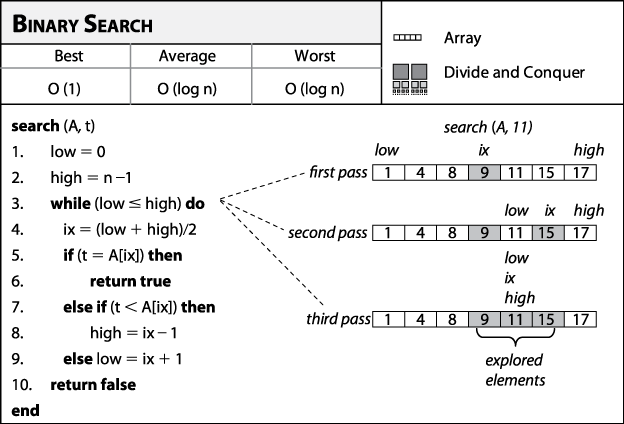
|  |  |  |
| --- | --- | --- |
|  | **Linear Search** | **Binary Search** |
| 2nd iteration | n-1 | n/2 |
| 3rd iteration | n-2 | n/4 |

**Example:**

****

**Algorithm and Complexity:**





**Best Case:**

Key is first compared with the middle element of the array.

The key is in the middle position of the array, the algorithm does only one comparison, irrespective of the size of the array.

T(n)=1

**Worst Case:**

In each iteration search space of BS is reduced by half, Maximum log n(base 2) array divisions are possible.

Recurrence relation is

T(n)=T(n/2) +1

Running Time is O(logn).

**Average Case:**

Key element neither is in the middle nor at the leaf level of the search tree.

It does half of the log n(base 2).

Base case=O(1)

Average and worst case=O(logn)

**Implementation:**

**Conclusion:**

|  |
| --- |
| Experiment No. 5 |
| Fractional Knapsack using Greedy Method |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 5**

**Title:** Fraction Knapsack

**Aim:** To study and implement Fraction Knapsack Algorithm

**Objective:** To introduce Greedy based algorithms

**Theory:**

Greedy method or technique is used to solve Optimization problems. A solution that can be maximized or minimized is called Optimal Solution.

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed size knapsack and must fill it with the most valuable items. The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one.

In Knapsack problem we are given:1) n objects 2) Knapsack with capacity m, 3) An object i is associated with profit Wi , 4) An object i is associated with profit Pi , 5) when an object i is placed in knapsack we get profit Pi Xi .

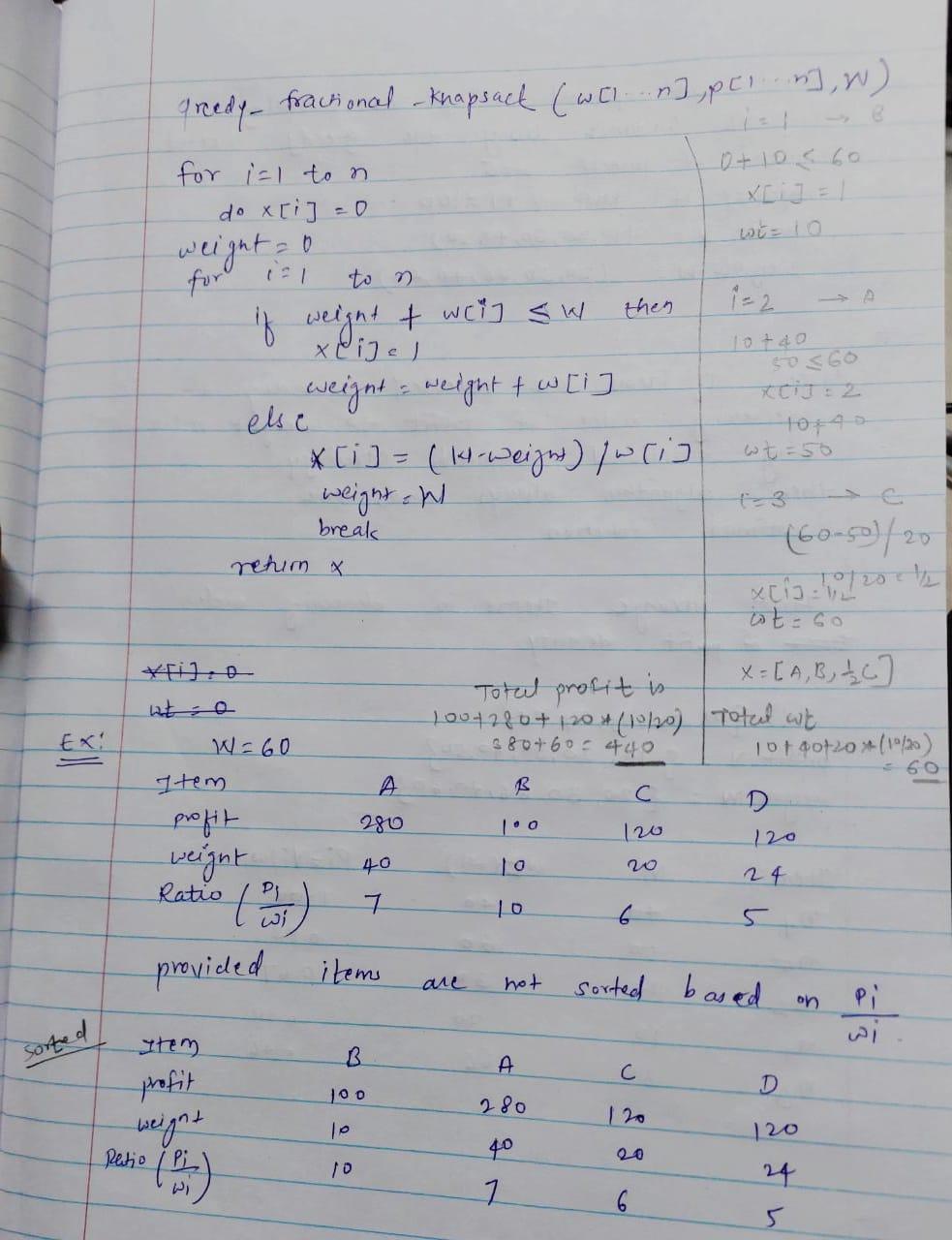
Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

**Example:**

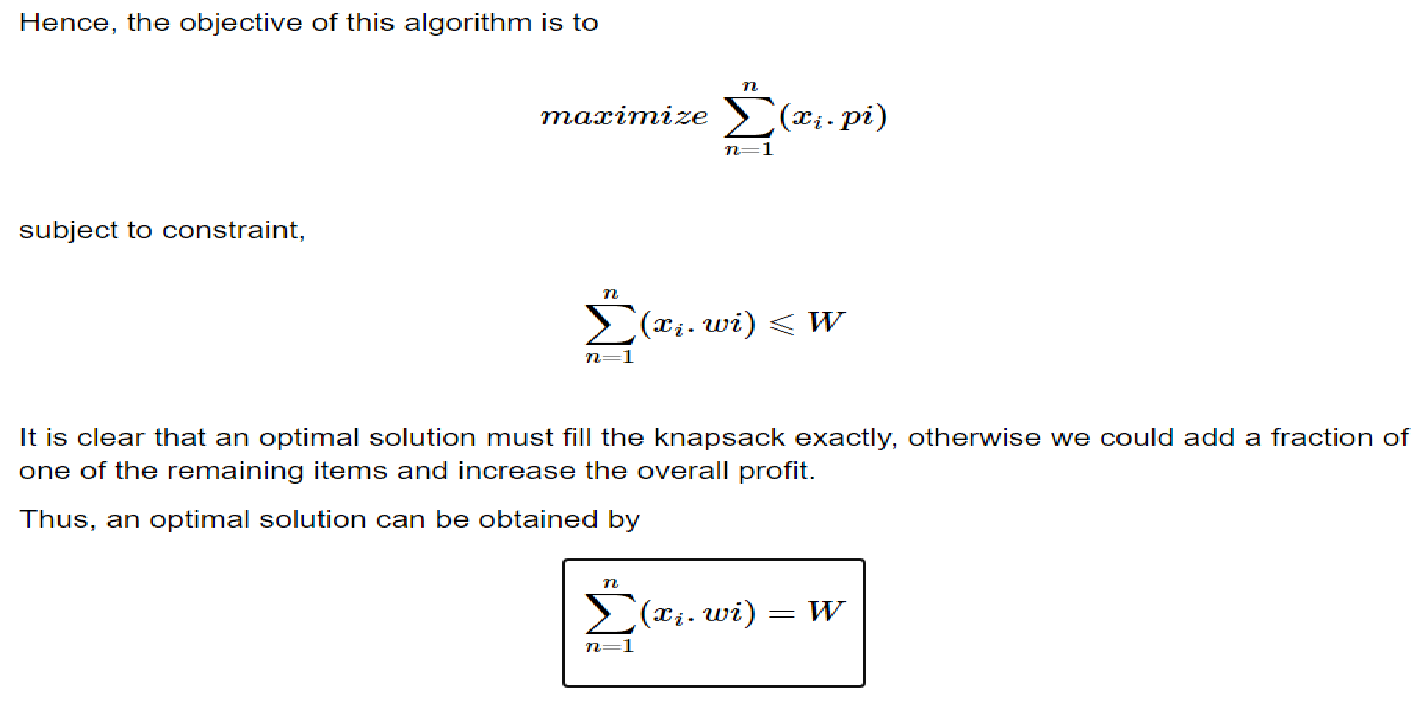
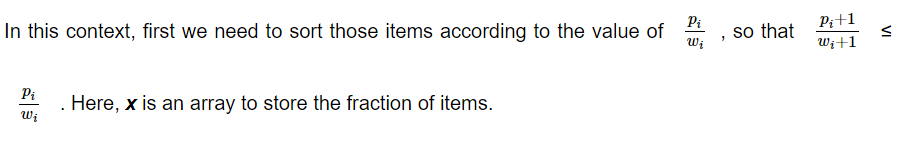
In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction *xi* of ith item.

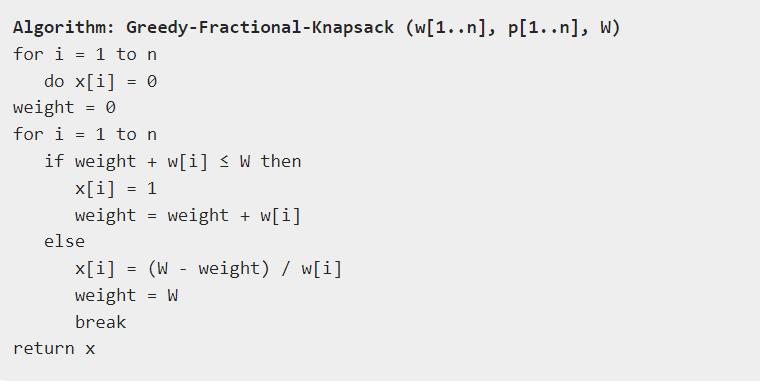
0⩽xi⩽1

The ith item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit**.**

****

**Algorithm:**

****



**Implementation:**

**Conclusion:** Fractional Knapsack algorithm has been successfully implemented.

|  |
| --- |
| Experiment No. 6 |
| Prim’s Algorithm |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 6**

**Title:** Prim’s Algorithm.

**Aim:** To study and implement Prim’s Minimum Cost Spanning Tree Algorithm.

**Objective:** To introduce Greedy based algorithms

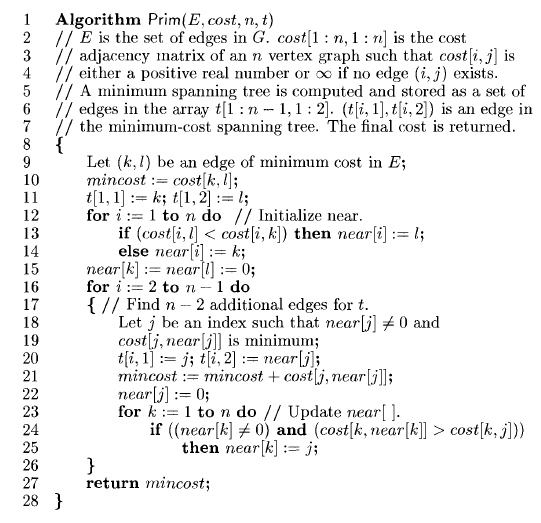
**Theory:**

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

**Example:**



**Algorithm and Complexity:**

****

Time Complexity is O( n2 ), Where, n = number of vertices **Theory:**

**Implementation:**

**Conclusion:** Comment on implementation of Prim’s Algorithm

|  |
| --- |
| Experiment No. 7 |
| Kruskal’s Algorithm |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 7**

**Title:** Kruskal’s Algorithm.

**Aim:** To study and implement Kruskal’s Minimum Cost Spanning Tree Algorithm.

**Objective:** To introduce Greedy based algorithms

**Theory:**

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

**Example:**



**Algorithm and Complexity:**

**A screenshot of a computer program

Description automatically generated**

Time Complexity is O(nlog n), Where, n = number of Edges

**Implementation:**

**Conclusion:** Comment on the implementation of Kruskal’s Algorithm.

|  |
| --- |
| Experiment No. 8 |
| Single Source Shortest Path using Dynamic Programming  (Bellman-Ford Algorithm) |
| Date of Performance: |
| Date of Submission: |

**Experiment No: 8**

**Title:** Single Source Shortest Path: Bellman Ford

**Aim:** To study and implement Single Source Shortest Path using Dynamic Programming: Bellman Ford

**Objective:** To introduceBellman Ford method

**Theory:**

Given a graph and a source vertex source in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.We have discussed Dijkstra’s algorithm for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

**Example:**

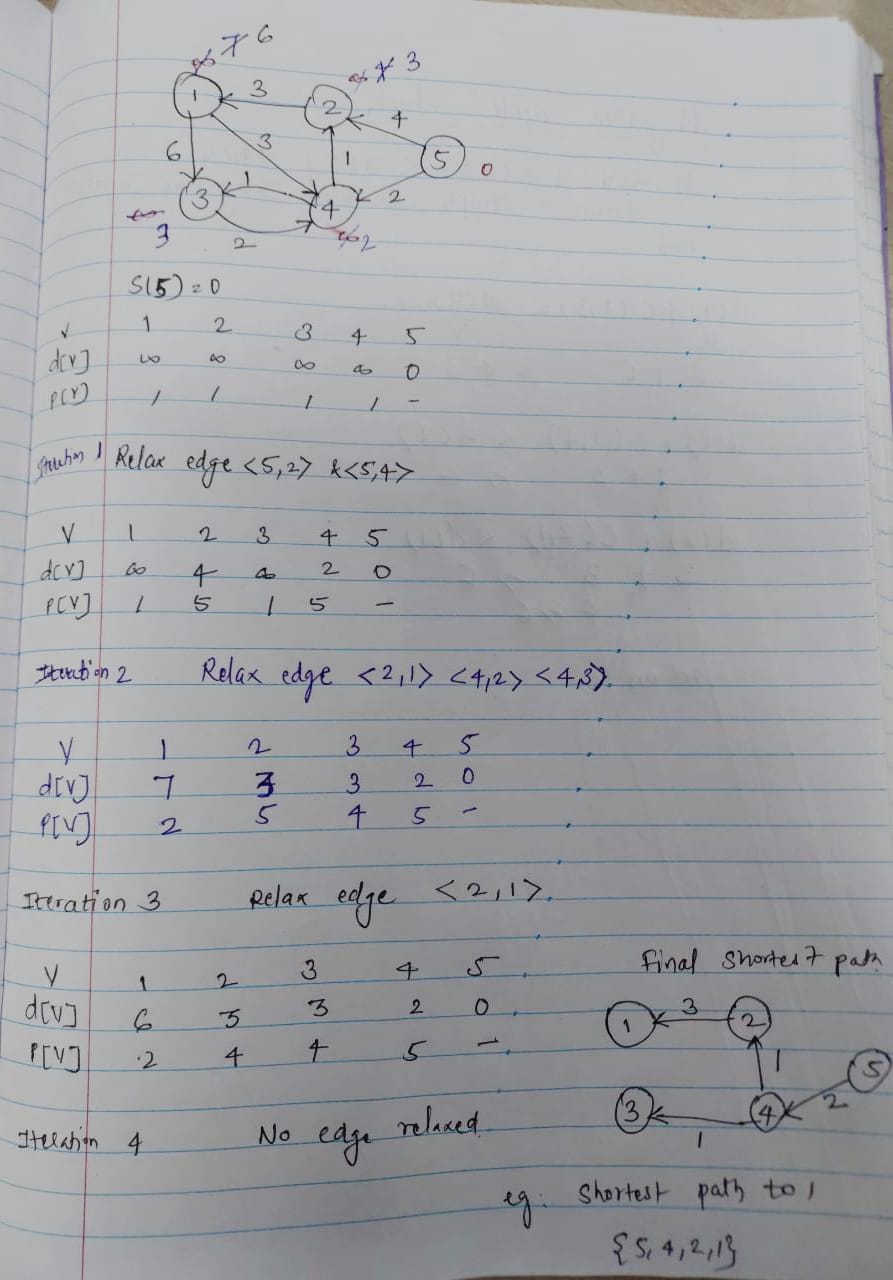
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

Bellman–Ford Algorithm Example Graph 1

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

Bellman–Ford Algorithm Example Graph 3

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.



**Algorithm:**

function Bellman\_Ford(list vertices, list edges, vertex source, distance[], parent[])  
   
// Step 1 – initialize the graph. In the beginning, all vertices weight of  
// INFINITY and a null parent, except for the source, where the weight is 0  
   
for each vertex v in vertices  
    distance[v] = INFINITY  
    parent[v] = NULL  
   
distance[source] = 0  
// Step 2 – relax edges repeatedly  
    for i = 1 to V-1    // V – number of vertices  
        for each edge (u, v) with weight w  
            if (distance[u] + w) is less than distance[v]  
                distance[v] = distance[u] + w  
                parent[v] = u  
   
// Step 3 – check for negative-weight cycles  
for each edge (u, v) with weight w  
    if (distance[u] + w) is less than distance[v]  
        return “Graph contains a negative-weight cycle”  
   
return distance[], parent[]

**Output:**

Shortest path from source (5)

Vertex 5 -> cost=0 parent=0

Vertex 1-> cost=6 parent=2

Vertex 2-> cost=3 parent=4

Vertex 3-> cost =3 parent =4

Vertex 4-> cost =2 paren=5

**Implementation:**

**Conclusion:** Comment on the implementation of Single Source Shortest Path: Bellman Ford algorithm.

|  |
| --- |
| Experiment No. 9 |
| Travelling Salesperson Problem using Dynamic Approach |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 9**

**Title:** Travelling Salesman Problem

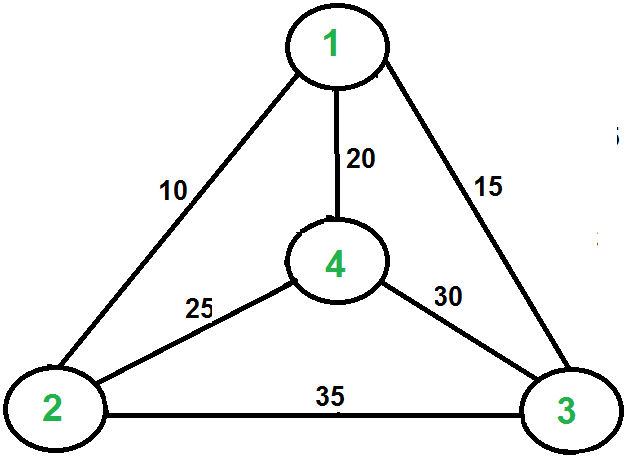
**Aim:** To study and implement Travelling Salesman Problem.

**Objective:** To introduce Dynamic Programming approach

**Theory:**

The **Traveling Salesman Problem (TSP)** is a classic optimization problem in which a salesperson needs to visit a set of cities exactly once and return to the starting city while minimizing the total distance traveled.

Given a set of cities and the distance between every pair of cities, find the **shortest possible route** that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hardproblem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

**Naive Solution:**

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! [Permutations](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)of cities.

3) Calculate the cost of every permutation and keep track of the minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: ?(n!)

**Dynamic Programming:**

Let the given set of vertices be {1, 2, 3, 4,.n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*. We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

**Implementation:**

**Conclusion:** Travelling Salesman Problem has been successfully implemented.

|  |
| --- |
| Experiment No. 10 |
| Sum of Subset using Backtracking |
| Date of Performance: |
| Date of Submission: |

**Title:** Sum of Subset

**Aim:** To study and implement Sum of Subset problem

**Objective:** To introduce Backtracking methods

**Theory:**

**Backtracking** is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn’t, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

* start with a sub-solution
* check if this sub-solution will lead to the solution or not
* If not, then come back and change the sub-solution and continue again.

The subset sum problem is a classic optimization problem that involves finding a subset of a given set of positive integers whose sum matches a given target value. More formally, given a set of non-negative integers and a target sum, we aim to determine whether there exists a subset of the integers whose sum equals the target.

Let's consider an example to better understand the problem. Suppose we have a set of integers [1, 4, 6, 8, 2] and a target sum of 9. We need to determine whether there exists a subset within the given set whose sum equals the target, in this case, 9. In this example, the subset [1, 8] satisfies the condition, as their sum is indeed 9.

**Solving Subset Sum with Backtracking**

To solve the subset, sum problem using backtracking, we will follow a recursive approach. Here's an outline of the algorithm:

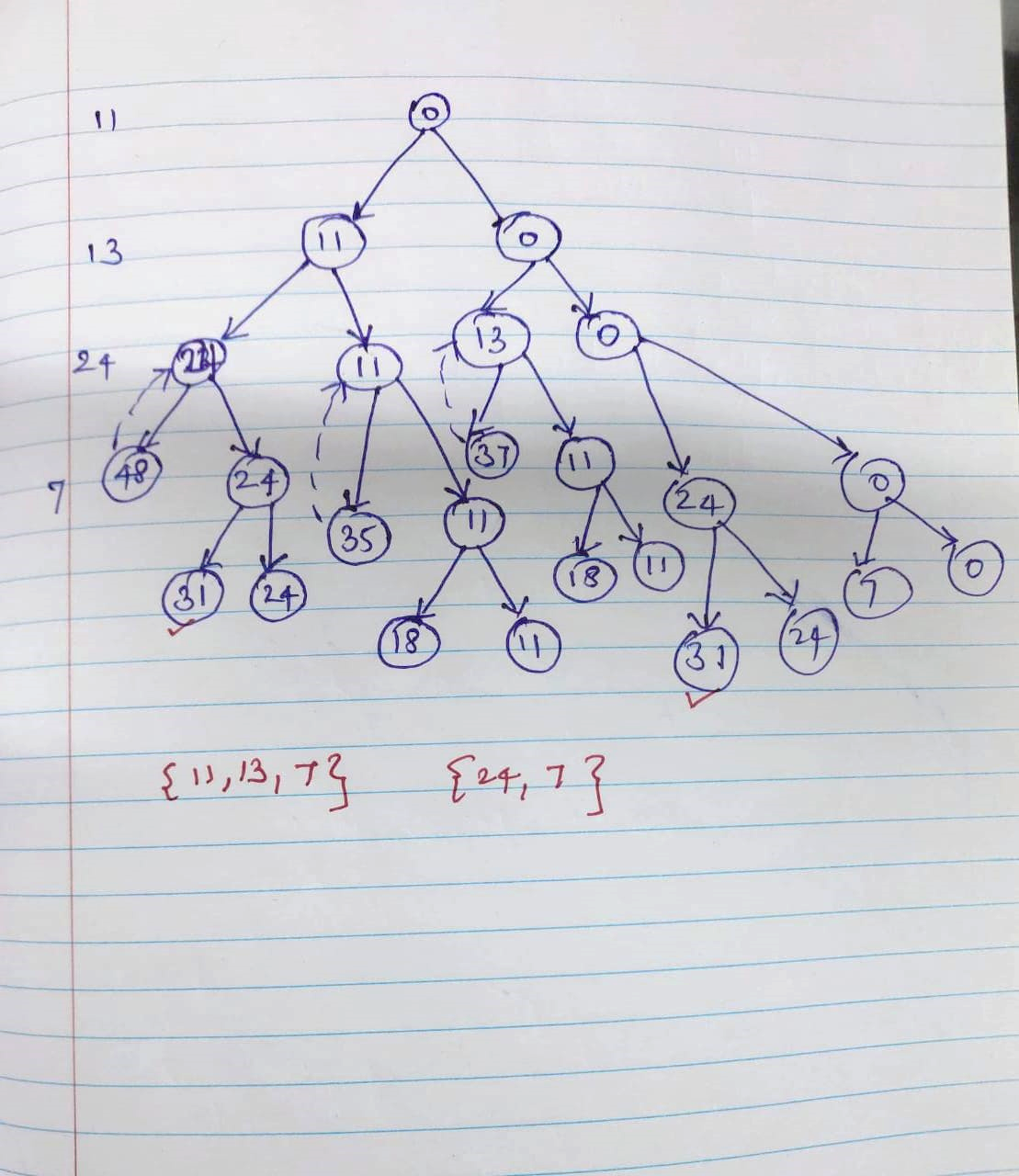
1. Sort the given set of integers in non-decreasing order.

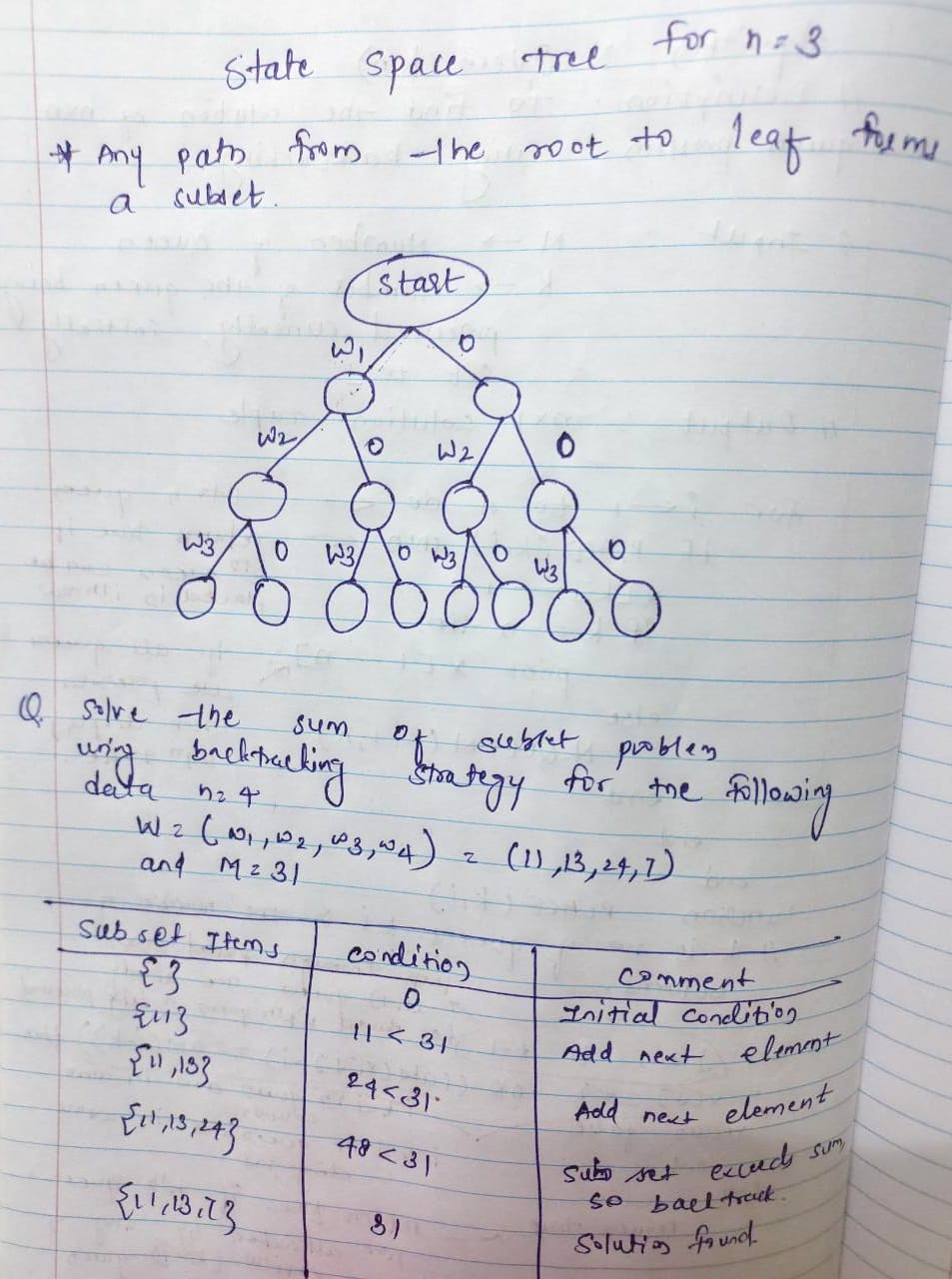
2. Start with an empty subset and initialize the current sum as 0.

3. Iterate through each integer in the set:

* Include the current integer in the subset.
* Increment the current sum by the value of the current integer.
* Recursively call the algorithm with the updated subset and current sum.
* If the current sum equals the target sum, we have found a valid subset.
* Backtrack by excluding the current integer from the subset.
* Decrement the current sum by the value of the current integer.

1. If we have exhausted all the integers and none of the subsets sum up to the target, we conclude that there is no valid subset.





**Implementation:**

**Conclusion:** The Sum of Subset problem has been implemented.

|  |
| --- |
| Experiment No. 11 |
| 15 puzzle problem |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 11**

**Title:** 15 Puzzle

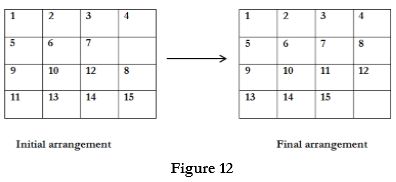
**Aim:** To study and implement 15 puzzle problem

**Objective:** To introduce Backtracking and Branch-Bound methods

**Theory:**

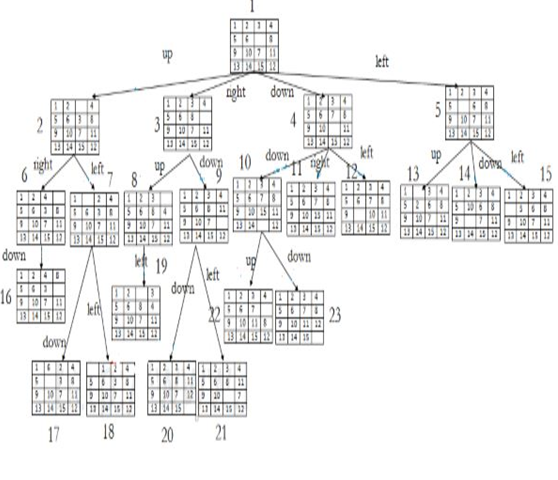
The 15 puzzle problem is invented by Sam Loyd in 1878.

* In this problem there are 15 tiles, which are numbered from 0 – 15.
* The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.
* The initial and goal arrangement is shown by following figure.



* There is always an empty slot in the initial arrangement.
* The legal moves are the moves in which the tiles adjacent to ES are moved to either left, right, up or down.
* Each move creates a new arrangement in a tile.
* These arrangements are called as states of the puzzle.
* The initial arrangement is called as initial state and goal arrangement is called as goal state.
* The state space tree for 15 puzzle is very large because there can be 16! Different arrangements.
* A partial state space tree can be shown in figure.
* In state space tree, the nodes are numbered as per the level.
* Each next move is generated based on empty slot positions.
* Edges are label according to the direction in which the empty space moves.
* The root node becomes the E – node.
* The child node 2, 3, 4 and 5 of this E – node get generated.
* Out of which node 4 becomes an E – node. For this node the live nodes 10, 11, 12 gets generated.
* Then the node 10 becomes the E – node for which the child nodes 22 and 23 gets generated.
* Finally we get a goal state at node 23.
* We can decide which node to become an E – node based on estimation formula.

**Example:**

****

**Implementation:**

**Conclusion:** The 15 Puzzle problem has been implemented.

|  |
| --- |
| Experiment No. 12 |
| Naïve String matching |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 12**

**Title:** Naïve String matching

**Aim:** To study and implement Naïve string matching Algorithm

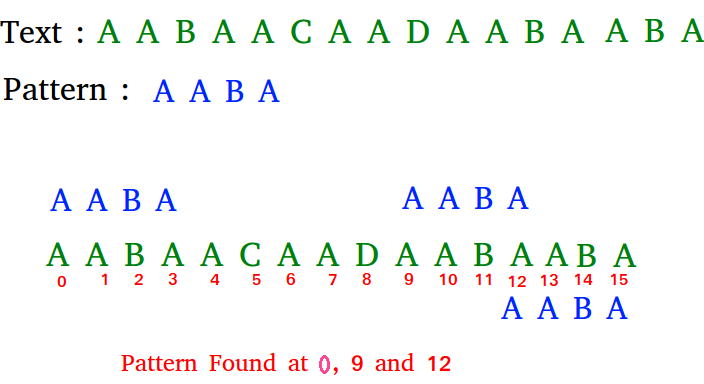
**Objective:** To introduce String matching methods

**Theory:**

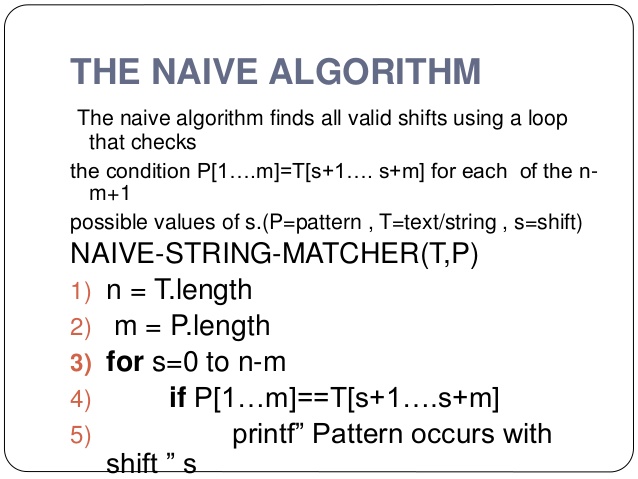
The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

**Example:**



**Algorithm:**



**Implementation:**

**Conclusion:** Comment on implementation of Naïve String-Matching algorithm.