

**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 | | |

**A black and white logo

AI-generated content may be incorrect.**

**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 analysing 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 and Merge 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 All Pairs Shortest Path Algorithm using Dynamic Method (Floyd Warshall). | - | - | - | 3 | - | - |
| 9. | To implement N Queen problem using Backtracking method. | - | - | - | - | 3 | - |
| 10. | 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 and Merge 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 All Pairs Shortest Path Algorithm using Dynamic Method (Floyd Warshall). |  |  |  |  |
| 9. | To implement N queen problem using Backtracking method. |  |  |  |  |
| 10. | Implement the Naïve string-matching algorithm and analyze 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 |
| --- |
| name Of The Student:-Bhagyashri Kaleni Sutar |
| Experiment Name :-Insertion sort |
| Date of Performance: |
| Date of Submission: |

**Title**: Insertion Sort

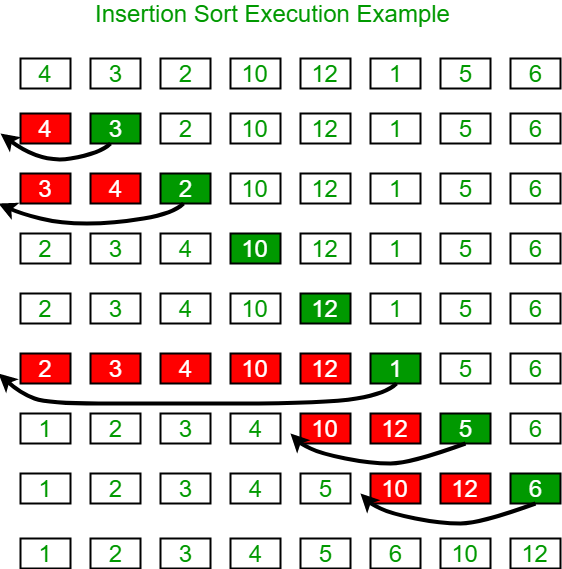
**Aim**: To implement Selection Comparative analysis for large values of ‘n’

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

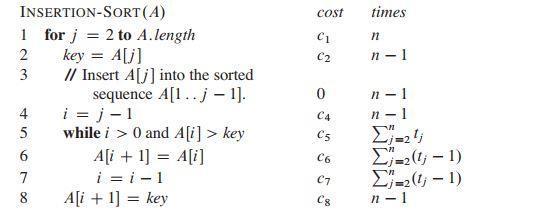
**Theory**:

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:**

**public class InsertionSort {**

**// Method to perform insertion sort**

**public static void insertionSort(int[] arr) {**

**// Iterate through the array starting from the second element**

**for (int i = 1; i < arr.length; i++) {**

**int key = arr[i]; // The element to be inserted into the sorted portion of the array**

**int j = i - 1;**

**// Shift elements of the sorted portion that are greater than the key**

**while (j >= 0 && arr[j] > key) {**

**arr[j + 1] = arr[j];**

**j = j - 1;**

**}**

**// Insert the key at the correct position**

**arr[j + 1] = key;**

**}**

**}**

**// Method to print the array**

**public static void printArray(int[] arr) {**

**for (int i : arr) {**

**System.out.print(i + " ");**

**}**

**System.out.println();**

**}**

**// Driver method**

**public static void main(String[] args) {**

**int[] arr = {12, 11, 13, 5, 6};**

**System.out.println("Original Array:");**

**printArray(arr);**

**// Call insertionSort to sort the array**

**insertionSort(arr);**

**System.out.println("Sorted Array:");**

**printArray(arr);**

**}**

**}**

**Output;-**

**Original Array:**

**12 11 13 5 6**

**Sorted Array:**

**5 6 11 12 13**

**Conclusion:**

**Insertion Sort is a simple and intuitive sorting algorithm that builds the sorted array one element at a time by repeatedly inserting each element into its correct position within the sorted portion of the array.**

| Experiment No.2 |
| --- |
| Name Of Student:-Bhagyashri Kaleni Sutar |
| Experiment Name:-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

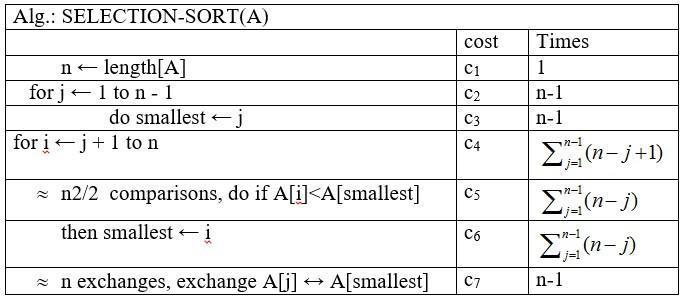
// Find the minimum element in arr[2...4] // and place it at beginning of arr[2...4]

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:**

**public class SelectionSort {**

**// Method to perform selection sort**

**public static void selectionSort(int[] arr) {**

**int n = arr.length;**

**// One by one move the boundary of unsorted subarray**

**for (int i = 0; i < n - 1; i++) {**

**// Find the minimum element in the unsorted portion**

**int minIndex = i;**

**for (int j = i + 1; j < n; j++) {**

**if (arr[j] < arr[minIndex]) {**

**minIndex = j;**

**}**

**}**

**// Swap the found minimum element with the first element of the unsorted portion**

**int temp = arr[minIndex];**

**arr[minIndex] = arr[i];**

**arr[i] = temp;**

**}**

**}**

**// Method to print the array**

**public static void printArray(int[] arr) {**

**for (int i : arr) {**

**System.out.print(i + " ");**

**}**

**System.out.println();**

**}**

**// Driver method**

**public static void main(String[] args) {**

**int[] arr = {64, 25, 12, 22, 11};**

**System.out.println("Original Array:");**

**printArray(arr);**

**// Call selectionSort to sort the array**

**selectionSort(arr);**

**System.out.println("Sorted Array:");**

**printArray(arr);**

**}**

**}**

**//output**

**Original Array:**

**64 25 12 22 11**

**Sorted Array:**

**11 12 22 25 64**

**Conclusion:**

**Selection Sort** is a simple and intuitive sorting algorithm that works by repeatedly selecting the smallest element from the unsorted portion of the array and swapping it with the first unsorted element. While easy to understand and implement, it has several limitations that make it inefficient for large datasets.

| Experiment No. 3 |
| --- |
| Name Of Student:-Bhagyashri Kaleni Sutar |
| Experiment Name:-Quick Sort and Merge sort |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 3**

**Title:** Quick Sort and Merge Sort

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

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

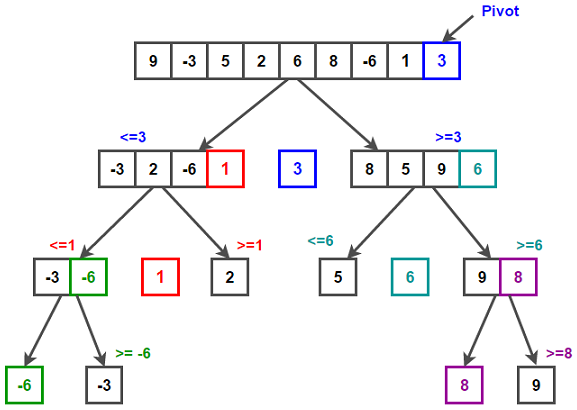
List : 3 7 8 5 2 1 9 5 4

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]

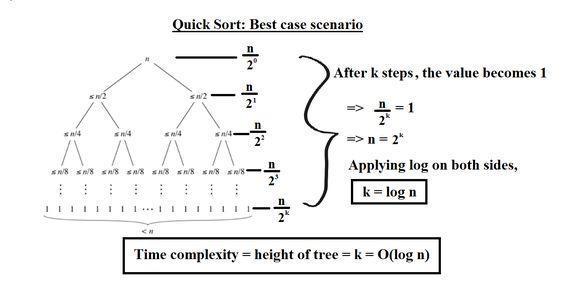
}

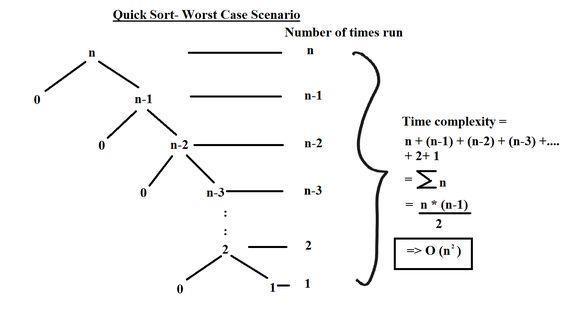
}

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

return (i + 1)

}



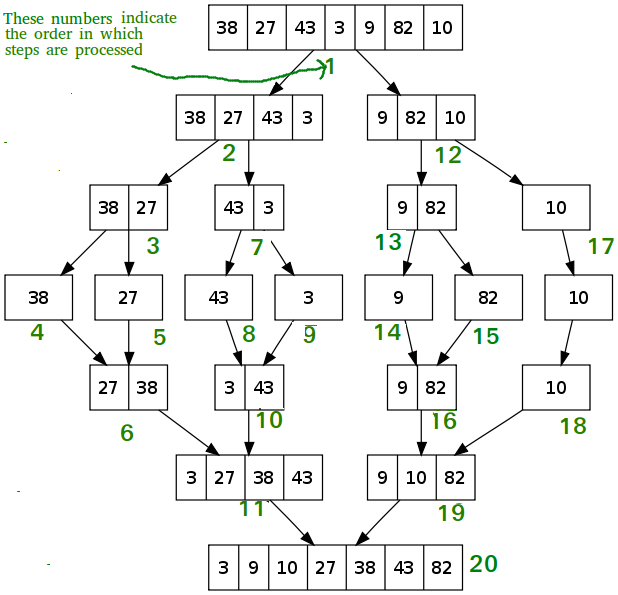


During the Mergesort process the object in the collection are divided into two collections. To split a collection, Mergesort will take the middle of the collection and split the collection into its left and its right part. The resulting collections are again recursively sorted via the Mergesort algorithm.

Once the sorting process of the two collections is finished, the result of the two collections is combined. To combine both collections Mergesort start at each collection at the beginning. It pick the object which is smaller and inserts this object into the new collection. For this collection it now selects the next elements and selects the smaller element from both collection.

Once all elements from both collections have been inserted in the new collection, Mergesort has successfully sorted the collection. To avoid the creation of too many collections, typically one new collection is created and the left and right side are treated as different collections.

**Example:**

****

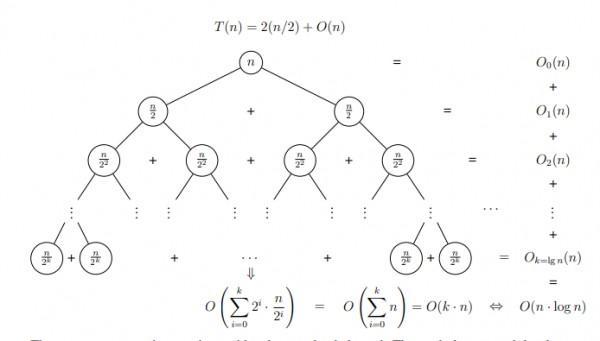
**Algorithm and Complexity:**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**A black and white text with white text

AI-generated content may be incorrect.**

****

**Implementation:**

**1.Quick Sort:-**

**import java.util.\*;**

**public class QuickSort {**

**public static void quickSort(int[] array, int low, int high) {**

**if (low < high) {**

**int pi = partition(array, low, high);**

**quickSort(array, low, pi - 1);**

**quickSort(array, pi + 1, high);**

**}**

**}**

**public static int partition(int[] array, int low, int high) {**

**int pivot = array[high]; // Taking the last element as pivot**

**int i = (low - 1); // Index of smaller element**

**// Traverse through all elements, compare each with pivot**

**for (int j = low; j < high; j++) {**

**if (array[j] < pivot) {**

**i++;**

**int temp = array[i];**

**array[i] = array[j];**

**array[j] = temp;**

**}**

**}**

**// Swap the pivot element with the element at (i + 1)**

**int temp = array[i + 1];**

**array[i + 1] = array[high];**

**array[high] = temp;**

**return i + 1; // Return the partition index**

**}**

**public static void printArray(int[] array) {**

**for (int i = 0; i < array.length; i++) {**

**System.out.print(array[i] + " ");**

**}**

**System.out.println();**

**}**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.in); // Corrected Scanner initialization**

**System.out.print("Enter the number of elements: ");**

**int n = sc.nextInt(); // Corrected nextint() to nextInt()**

**int[] array = new int[n];**

**System.out.println("Enter " + n + " integers:");**

**for (int i = 0; i < n; i++) {**

**array[i] = sc.nextInt();**

**}**

**System.out.println("Original array:");**

**printArray(array);**

**// Sorting the array using QuickSort**

**quickSort(array, 0, array.length - 1);**

**System.out.println("Sorted array:");**

**printArray(array);**

**sc.close(); // Close the scanner after use**

**}**

**}**

**Output:-**

**Enter the number of elements: 10**

**Enter 10 integers:**

**2**

**4**

**1**

**9**

**8**

**5**

**4**

**6**

**7**

**8**

**Original array:**

**2 4 1 9 8 5 4 6 7 8**

**conclusion:-**

**Pivot Selection: The pivot is always the last element in the array. This can cause slow performance (O(n²)) when the array is sorted or nearly sorted.**

**Time Complexity: On average, QuickSort runs in O(n log n) time. But in the worst case (like with a bad pivot), it can slow down to O(n²), especially if the array is already sorted.**

**Space Complexity: The space complexity is O(log n) due to the recursion stack. In the worst case, it could be O(n).**

**In-place Sorting: QuickSort sorts the array without needing extra space.**

**Efficiency: QuickSort is fast for large datasets, especially if the pivot selection is improved to avoid worst-case scenarios.**

**2.Merge Sort:-**

**// Java program for Merge Sort**

**class MergeSort**

**{**

**// Merges two subarrays of a[]**

**void merge(int a[], int l, int m, int r)**

**{**

**int n1 = m - l + 1;**

**int n2 = r - m;**

**int L[] = new int[n1];**

**int R[] = new int[n2];**

**for (int i = 0; i < n1; ++i)**

**L[i] = a[l + i];**

**for (int j = 0; j < n2; ++j)**

**R[j] = a[m + 1 + j];**

**// Merge the temp arrays**

**// Initial indexes of first and second subarrays**

**int i = 0, j = 0;**

**int k = l;**

**while (i < n1 && j < n2)**

**{**

**if (L[i] <= R[j]) {**

**a[k] = L[i];**

**i++;**

**}**

**else {**

**a[k] = R[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1)**

**{**

**a[k] = L[i];**

**i++;**

**k++;**

**}**

**while (j < n2)**

**{**

**a[k] = R[j];**

**j++;**

**k++;**

**}**

**}**

**// Main function that sorts a[l..r] using**

**// merge()**

**void sort(int a[], int l, int r)**

**{**

**if (l < r) {**

**int m = (l + r) / 2;**

**// Sort first and second halves**

**sort(a, l, m);**

**sort(a, m + 1, r);**

**// Merge the sorted halves**

**merge(a, l, m, r);**

**}**

**}**

**// Driver method**

**public static void main(String args[])**

**{**

**int a[] = {4,5,3,8,9,11,10 };**

**// Calling of Merge Sort**

**MergeSort ob = new MergeSort();**

**ob.sort(a, 0, a.length - 1);**

**int n = a.length;**

**for (int i = 0; i < n; ++i)**

**System.out.print(a[i] + " ");**

**}**

**}**

**Output:-**

**3 4 5 8 9 10 11**

**// \*\*Time Complexity Analysis of Merge Sort:\*\***

**// Consider the following terminologies:**

**// T(k) = time taken to sort k elements**

**// M(k) = time taken to merge k elements**

**// So, it can be written**

**// T(N) = 2 \* T(N/2) + M(N)**

**// = 2 \* T(N/2) + constant \* N**

**// These N/2 elements are further divided into two halves. So,**

**// T(N) = 2 \* [2 \* T(N/4) + constant \* N/2] + constant \* N**

**// = 4 \* T(N/4) + 2 \* N \* constant**

**// . . .**

**// = 2k \* T(N/2k) + k \* N \* constant**

**// It can be divided maximum until there is one element left. So, then N/2k = 1. k = log2N**

**// T(N) = N \* T(1) + N \* log2N \* constant**

**// = N + N \* log2N**

**// Therefore the time complexity is O(N \* log2N).**

**// So in the best case, the worst case and the average case the time complexity is the same.**

**// \*\*Space Complexity Analysis of Merge Sort:\*\***

**// Merge sort has a space complexity of O(n).**

**// This is because it uses an auxiliary array of size n to merge the sorted halves of the input array. The auxiliary array is used to store the merged result, and the input array is overwritten with the sorted result.**

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

### 1. Quick Sort

**Quick Sort is a divide-and-conquer algorithm that divides the array into smaller sub-arrays, then recursively sorts those sub-arrays. The key idea is to choose a "pivot" element from the array, partition the array into two sub-arrays (one with elements smaller than the pivot and one with elements larger than the pivot), and then recursively apply the same process to the sub-arrays.**

### 2. Merge Sort:-

**Merge Sort is another divide-and-conquer algorithm. It splits the array into halves, recursively sorts each half, and then merges the two sorted halves back together. Unlike Quick Sort, Merge Sort always divides the array into two halves, so it has a consistent performance, regardless of the input data.**

| Experiment No. 4 |
| --- |
| Name Of Student:-Bhagyashri Kaleni Sutar |
| Name Of Experiment:-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.

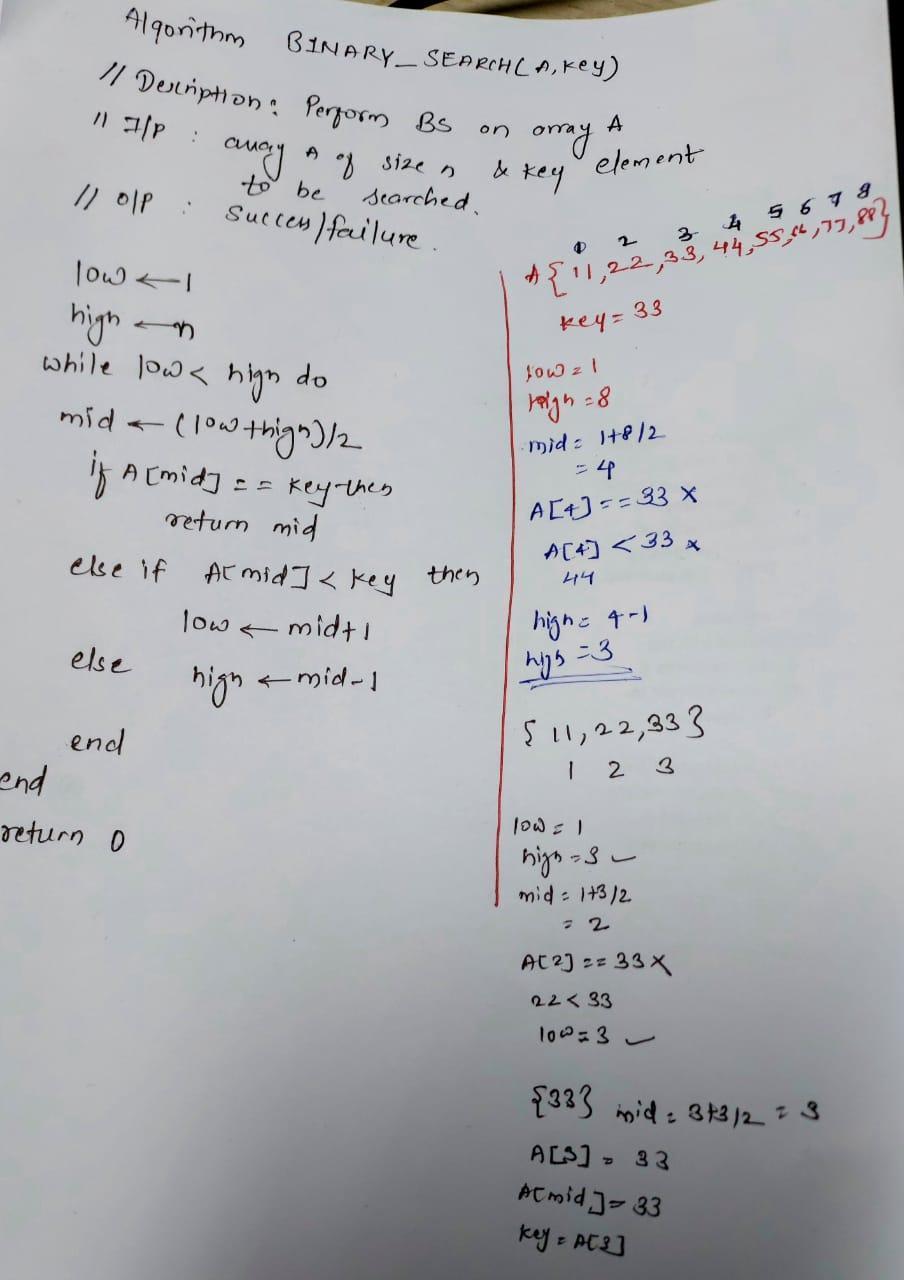


The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

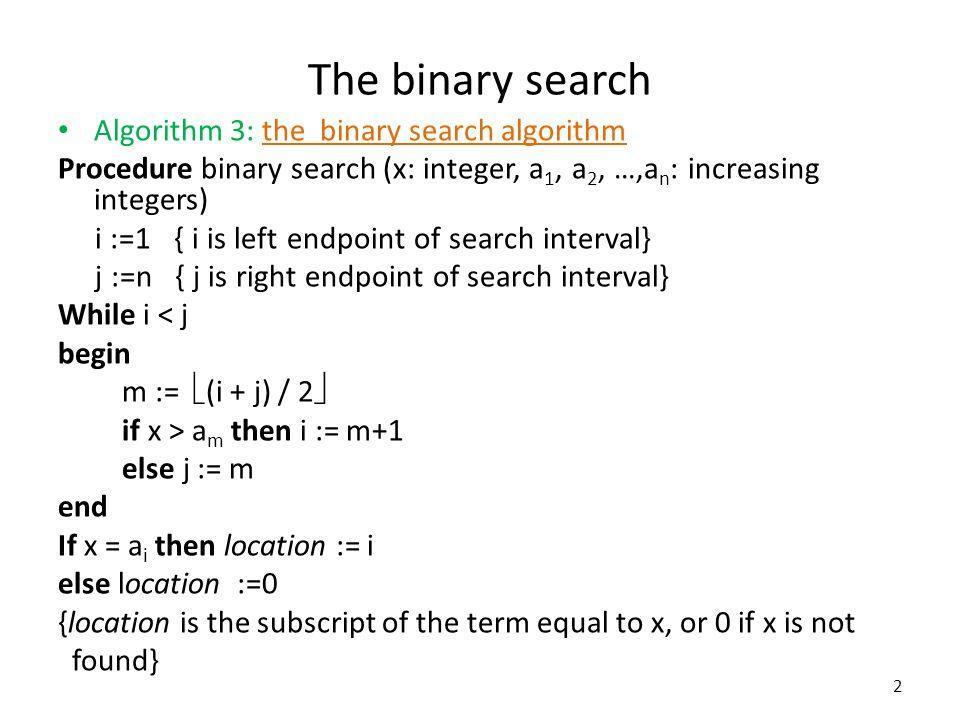
* Compare x with the middle element.
* If x matches with the middle element, we return the mid index.
* Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
* Else (x is smaller) recur for the left half.
* Binary 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.

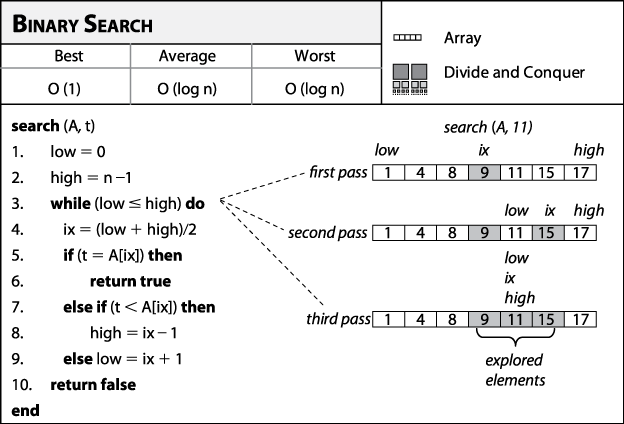
|  | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| 2nd iteration | n-1 | n/2 |
| 3rd iteration | n-2 | n/4 |

**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:**

**class BinarySearch {**

**int binarySearch(int a[], int l, int r, int x)**

**{**

**// Scanner sc=new Scanner(System.in);**

**// System.out.println("Enter Element Of array: ");**

**// int a[]=sc.nexta();**

**// System.out.println("Enter Element To be search: ");**

**// int x=sc.nextInt();**

**while (l <= r)**

**{**

**int m = (l + r) / 2;**

**if (a[m] == x)**

**{**

**return m;**

**// If element is smaller than mid, then**

**// it can only be present in left subarray**

**// so we decrease our r pointer to mid - 1**

**}**

**else if (a[m] > x)**

**{**

**r = m - 1;**

**// Else the element can only be present**

**// in right subarray**

**// so we increase our l pointer to mid + 1**

**}**

**else**

**{**

**l = m + 1;**

**}**

**}**

**// No Element Found**

**return -1;**

**}**

**public static void main(String args[])**

**{**

**BinarySearch ob = new BinarySearch();**

**int a[] = { 2, 3, 4, 10, 40 };**

**int n = a.length;**

**int x = 4;**

**int res = ob.binarySearch(a,0, n - 1, x);**

**if (res == -1)**

**System.out.println("Element not present");**

**else**

**System.out.println("Element found at index " + res);**

**}**

**}**

**Output:-**

**Element found at index 2**

**//\*\*Time Complexity\*\***

**1. Best Case (O(1)): In the best-case scenario, binary search finds the target value immediately at the mid-point of the array.**

**This requires only a single comparison, making the time complexity O(1).**

**The algorithm achieves its optimal performance when the target is perfectly aligned with the mid-point on the first attempt.**

**2. Average Case (O(log n)): On average, binary search performs O(log n) comparisons.**

**Each step halves the search range, leading to logarithmic time complexity.**

**This efficiency holds across various positions of the target within the sorted array, as the algorithm consistently reduces the problem size exponentially.**

**3. Worst Case (O(log n)): In the worst-case scenario, binary search also operates with O(log n) complexity.**

**This occurs when the target value is not present, requiring the algorithm to explore the entire search range.**

**Despite this, the logarithmic reduction of the search space ensures that the time complexity remains logarithmic.**

**\*\*Space Complexity \*\***

**1. O(1) (iterative version)**

**2.O(logn) (recursive version)**

**Conclusion:**

**Binary Search is one of the most efficient algorithms for finding an element in a sorted array or list. It significantly reduces the search time compared to a linear search, making it an essential tool in various applications where searching needs to be optimized.**

| Experiment No. 5 |
| --- |
| Name Of Student :-Bhagyashri Kaleni Sutar |
| Experiment Name:-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 .

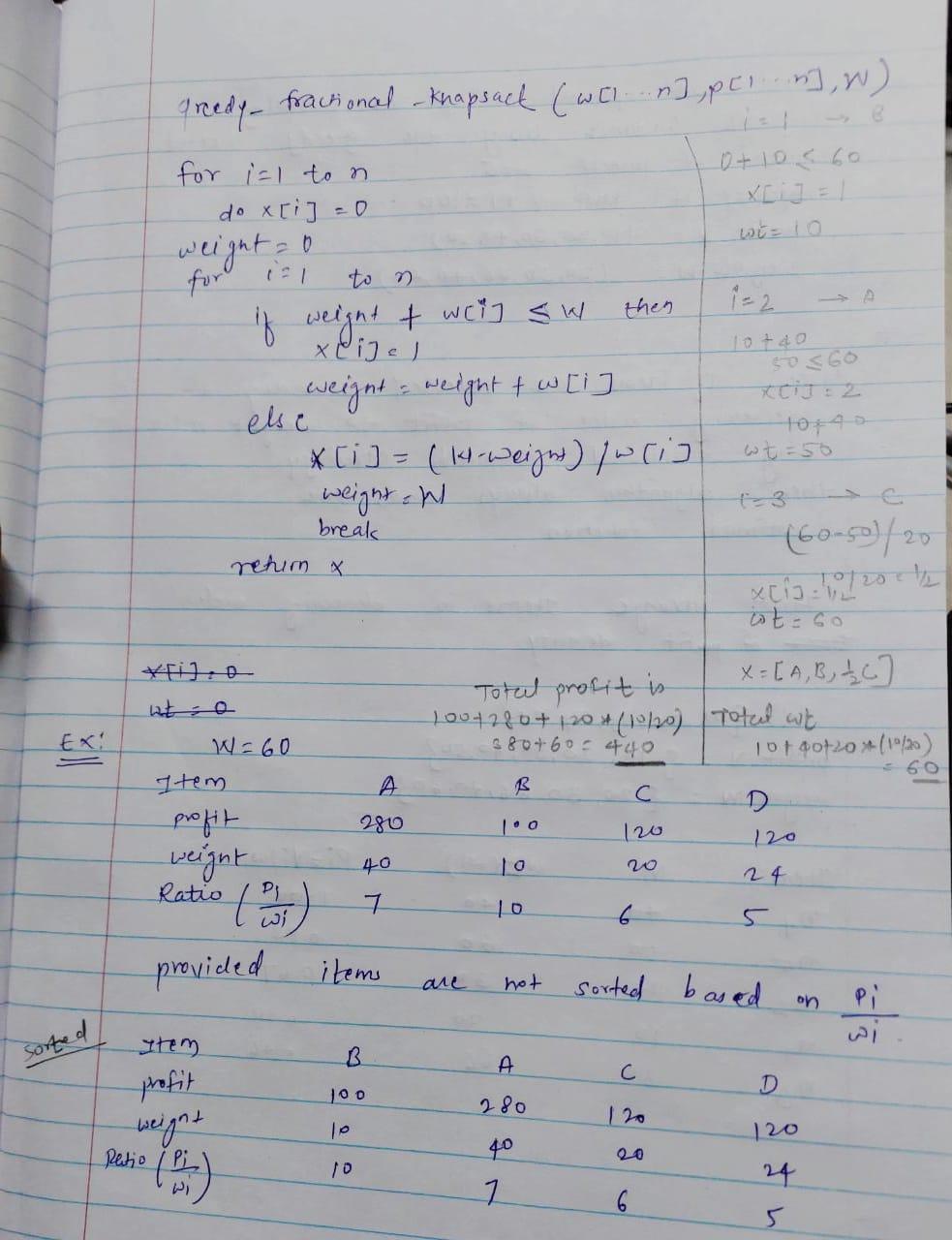
Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

**Example:**

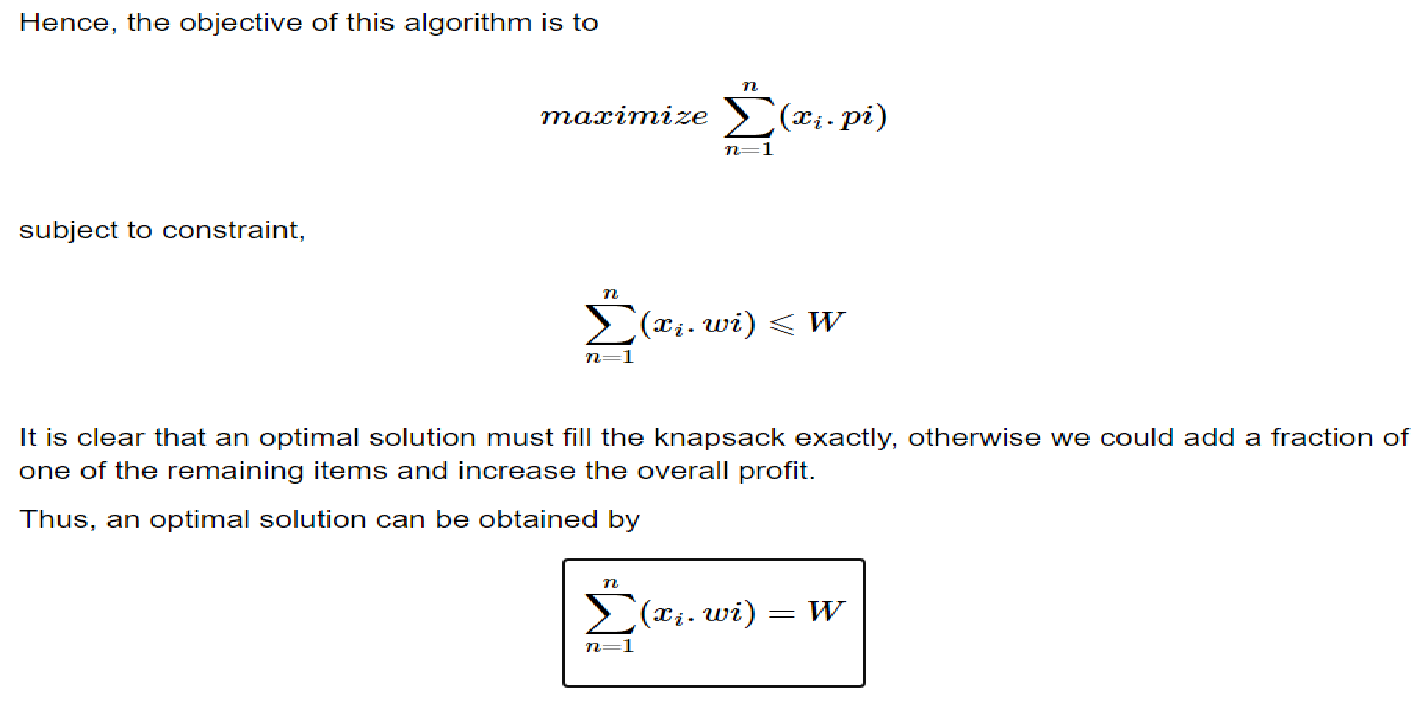
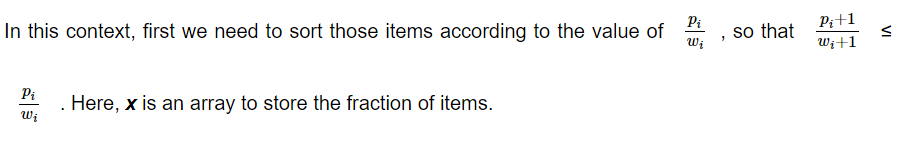
In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction *xi* of ith item.

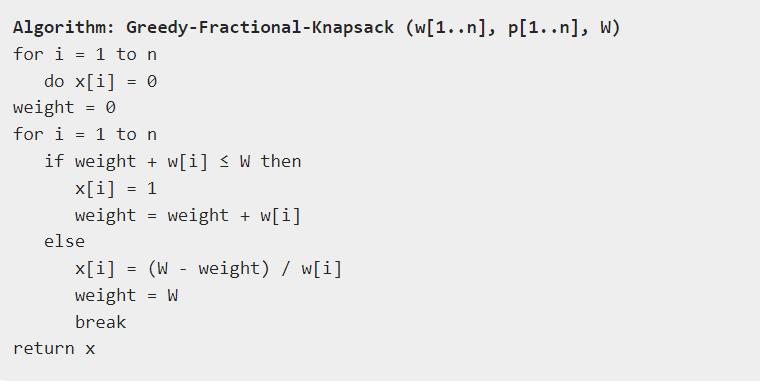
0⩽xi⩽1

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:**

**import java.util.Arrays;**

**import java.util.Comparator;**

**class Item**

**{**

**int weight;**

**int profit;**

**double ratio;**

**public Item(int weight, int value)**

**{**

**this.weight = weight;**

**this.profit = value;**

**this.ratio = (double) value / weight;**

**}**

**}**

**public class Knapsack {**

**public static double getMaxValue(Item[] items, int capacity)**

**{**

**Arrays.sort(items, new Comparator<Item>()**

**{**

**public int compare(Item a, Item b)**

**{**

**return Double.compare(b.ratio, a.ratio);**

**}**

**});**

**double totalValue = 0.0;**

**for (Item item : items) {**

**if (capacity == 0) {**

**break;**

**}**

**if (item.weight <= capacity)**

**{**

**totalValue += item.profit;**

**capacity -= item.weight;**

**}**

**else**

**{**

**totalValue += item.profit \* ((double) capacity / item.weight);**

**break;**

**}**

**}**

**return totalValue;**

**}**

**public static void main(String[] args)**

**{**

**int[] profits = {10, 20, 30, 40, 50};**

**int[] weights = {20, 30, 66, 40, 60};**

**Item[] items = new Item[profits.length];**

**for (int i = 0; i < profits.length; i++)**

**{**

**items[i] = new Item(weights[i], profits[i]);**

**}**

**int capacity = 100;**

**double maxValue = getMaxValue(items, capacity);**

**System.out.println("Maximum value in Knapsack: " + maxValue);**

**}**

**}**

**//Output:-**

**Maximum value in Knapsack: 90.0**

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

| Experiment No. 6 |
| --- |
| Name Of Student:-Bhagyashri Kaleni Sutar |
| Experiment Name:-Prim’s Algorithm |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 6**

**Title:** Prim’s Algorithm.

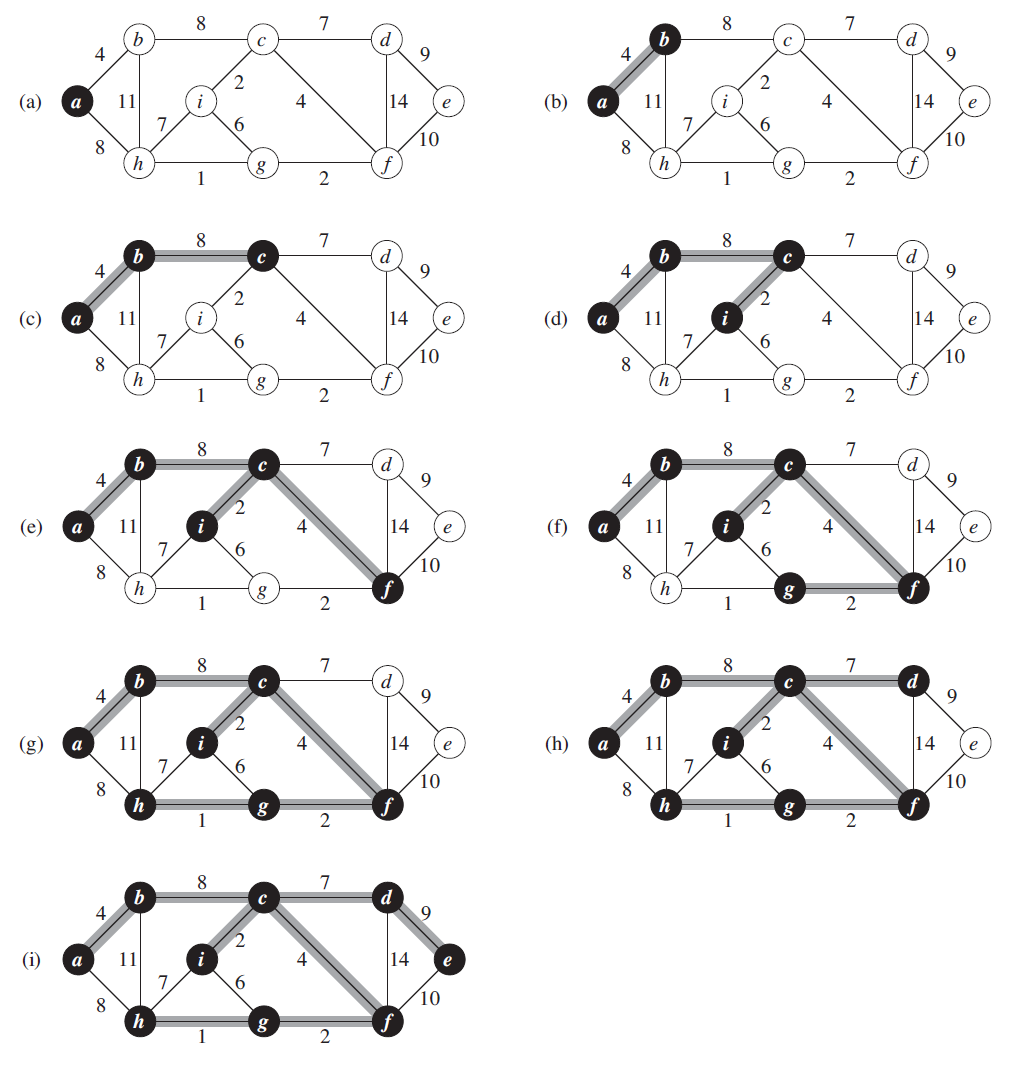
**Aim:** To study and implement Prim’s Minimum Cost Spanning Tree Algorithm.

**Objective:** To introduce Greedy based algorithms

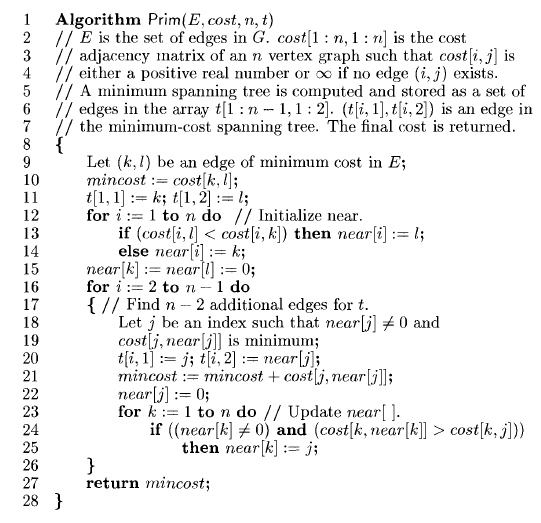
**Theory:**

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:**

**// A Java program for Prim's Minimum Spanning Tree (MST)**

**// algorithm. The program is for adjacency matrix**

**// representation of the graph**

**import java.io.\*;**

**import java.lang.\*;**

**import java.util.\*;**

**class PRIMS{**

**// A utility function to find the vertex with minimum**

**// key value, from the set of vertices not yet included**

**// in MST**

**int minKey(int key[], Boolean mstSet[])**

**{**

**// Initialize min value**

**int min = Integer.MAX\_VALUE, min\_index = -1;**

**for (int v = 0; v < mstSet.length; v++)**

**if (mstSet[v] == false && key[v] < min) {**

**min = key[v];**

**min\_index = v;**

**}**

**return min\_index;**

**}**

**// A utility function to print the constructed MST**

**// stored in parent[]**

**void printMST(int parent[], int graph[][])**

**{**

**System.out.println("Edge \tWeight");**

**for (int i = 1; i < graph.length; i++)**

**System.out.println(parent[i] + " - " + i + "\t"**

**+ graph[parent[i]][i]);**

**}**

**// Function to construct and print MST for a graph**

**// represented using adjacency matrix representation**

**void primMST(int graph[][])**

**{**

**int V = graph.length;**

**// Array to store constructed MST**

**int parent[] = new int[V];**

**// Key values used to pick minimum weight edge in**

**// cut**

**int key[] = new int[V];**

**// To represent set of vertices included in MST**

**Boolean mstSet[] = new Boolean[V];**

**// Initialize all keys as INFINITE**

**for (int i = 0; i < V; i++) {**

**key[i] = Integer.MAX\_VALUE;**

**mstSet[i] = false;**

**}**

**// Always include first 1st vertex in MST.**

**// Make key 0 so that this vertex is**

**// picked as first vertex**

**key[0] = 0;**

**// First node is always root of MST**

**parent[0] = -1;**

**// The MST will have V vertices**

**for (int count = 0; count < V - 1; count++) {**

**// Pick the minimum key vertex from the set of**

**// vertices not yet included in MST**

**int u = minKey(key, mstSet);**

**// Add the picked vertex to the MST Set**

**mstSet[u] = true;**

**// Update key value and parent index of the**

**// adjacent vertices of the picked vertex.**

**// Consider only those vertices which are not**

**// yet included in MST**

**for (int v = 0; v < V; v++)**

**// graph[u][v] is non zero only for adjacent**

**// vertices of m mstSet[v] is false for**

**// vertices not yet included in MST Update**

**// the key only if graph[u][v] is smaller**

**// than key[v]**

**if (graph[u][v] != 0 && mstSet[v] == false**

**&& graph[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = graph[u][v];**

**}**

**}**

**// Print the constructed MST**

**printMST(parent, graph);**

**}**

**public static void main(String[] args)**

**{**

**PRIMS t = new PRIMS();**

**int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },**

**{ 2, 0, 3, 8, 5 },**

**{ 0, 3, 0, 0, 7 },**

**{ 6, 8, 0, 0, 9 },**

**{ 0, 5, 7, 9, 0 } };**

**// Print the solution**

**t.primMST(graph);**

**}**

**}**

**Output:-**

**Edge Weight**

**0 - 1 2**

**1 - 2 3**

**0 - 3 6**

**1 - 4 5**

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

**Prim's Algorithm** is a greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a connected, undirected graph with weighted edges. The goal is to select the edges that connect all the vertices of the graph while minimizing the total edge weight, ensuring that no cycles are formed.

| Experiment No. 7 |
| --- |
| name Of Student:-Bhagyashri kaleni Sutar |
| Experiment Name:-Kruskal’s Algorithm |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 7**

**Title:** Kruskal’s Algorithm.

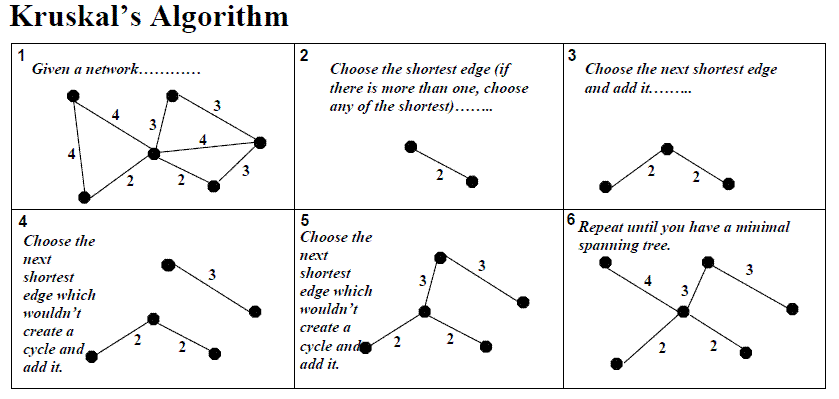
**Aim:** To study and implement Kruskal’s Minimum Cost Spanning Tree Algorithm.

**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:**

**import java.util.Arrays;**

**import java.util.Comparator;**

**class Kruskal {**

**public static int kruskalsMST(int V, int[][] edges) {**

**// Sort all edges based on weight**

**Arrays.sort(edges, Comparator.comparingInt(e -> e[2]));**

**// Traverse edges in sorted order**

**DSU dsu = new DSU(V);**

**int cost = 0, count = 0;**

**for (int[] e : edges) {**

**int x = e[0], y = e[1], w = e[2];**

**// Make sure that there is no cycle**

**if (dsu.find(x) != dsu.find(y)) {**

**dsu.union(x, y);**

**cost += w;**

**if (++count == V - 1) break;**

**}**

**}**

**return cost;**

**}**

**public static void main(String[] args) {**

**// An edge contains, weight, source and destination**

**int[][] edges = {**

**{0, 1, 10}, {1, 3, 15}, {2, 3, 4}, {2, 0, 6}, {0, 3, 5}**

**};**

**System.out.println(kruskalsMST(4, edges));**

**}**

**}**

**// Disjoint set data structure**

**class DSU {**

**private int[] parent, rank;**

**public DSU(int n) {**

**parent = new int[n];**

**rank = new int[n];**

**for (int i = 0; i < n; i++) {**

**parent[i] = i;**

**rank[i] = 1;**

**}**

**}**

**public int find(int i) {**

**if (parent[i] != i) {**

**parent[i] = find(parent[i]);**

**}**

**return parent[i];**

**}**

**public void union(int x, int y) {**

**int s1 = find(x);**

**int s2 = find(y);**

**if (s1 != s2) {**

**if (rank[s1] < rank[s2]) {**

**parent[s1] = s2;**

**} else if (rank[s1] > rank[s2]) {**

**parent[s2] = s1;**

**} else {**

**parent[s2] = s1;**

**rank[s1]++;**

**}**

**}**

**}**

**}**

**Output:-**

**19**

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

**Kruskal’s Algorithm** is a well-known greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a connected, undirected graph with weighted edges. Like Prim’s algorithm, Kruskal’s algorithm ensures that the selected edges have the minimum possible weight, and no cycles are formed while connecting all the vertices of the graph.

| Experiment No. 8 |
| --- |
| To implement All Pairs Shortest Path Algorithm using Dynamic Method (Floyd Warshall). |
| Date of Performance: |
| Date of Submission: |

**Experiment No: 8**

**Title:** All Pair Shortest Path: Floyd Warshall Algorithm

**Aim:** To study and implement All Pair Shortest Path using Dynamic Programming: Floyd Warshall

**Objective:** To introduceFloyd Warshall method

**Theory:**

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs can handle graphs with both positive and negative edge weights. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Suppose we have a graph G[][] with V vertices from 1 to N. Now we have to evaluate a shortestPathMatrix[][] where shortestPathMatrix[i][j] represents the shortest path between vertex i to j.

Obviously the shortest path between i to j will have some k number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from 1 to N as an intermediate node one by one.

A green rectangle with black arrows and white text

AI-generated content may be incorrect.

**Algorithm**

* Initialize the solution matrix same as the input graph matrix as a first step.
* Then update the solution matrix by considering all vertices as an intermediate vertex.
* The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
* When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}**as intermediate vertices.
* For every pair**(i, j)** of the source and destination vertices respectively, there are two possible cases.
  + **k** is not an intermediate vertex in shortest path from**i**to**j**. We keep the value of**dist[i][j]**as it is.
  + **k** is an intermediate vertex in shortest path from **i** to**j**. We update the value of**dist[i][j]**as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

For k = 0 to n – 1

For i = 0 to n – 1

For j = 0 to n – 1

Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])

where i = source Node, j = Destination Node, k = Intermediate Node

**Implementation:**

**public class FloydWarshall {**

**// Method to implement Floyd-Warshall algorithm**

**public static void floydWarshall(int[][] graph) {**

**int V = graph.length; // Number of vertices in the graph**

**// dist[][] will hold the shortest distance between every pair of vertices**

**int[][] dist = new int[V][V];**

**// Initialize the solution matrix same as the input graph matrix.**

**// dist[][] will be the output matrix that will eventually have the shortest distances**

**// between every pair of vertices.**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**if (i == j) {**

**dist[i][j] = 0; // Distance from a vertex to itself is always 0**

**} else if (graph[i][j] != 0) {**

**dist[i][j] = graph[i][j]; // Distance from i to j is the edge weight**

**} else {**

**dist[i][j] = Integer.MAX\_VALUE; // If no edge exists, set it as infinity**

**}**

**}**

**}**

**// Update the distance matrix by considering each vertex as an intermediate vertex.**

**for (int k = 0; k < V; k++) {**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**// If a shorter path from i to j through k is found, update dist[i][j]**

**if (dist[i][k] != Integer.MAX\_VALUE && dist[k][j] != Integer.MAX\_VALUE &&**

**dist[i][j] > dist[i][k] + dist[k][j]) {**

**dist[i][j] = dist[i][k] + dist[k][j];**

**}**

**}**

**}**

**}**

**// Print the shortest distance matrix**

**printSolution(dist);**

**}**

**// Method to print the solution**

**public static void printSolution(int[][] dist) {**

**int V = dist.length;**

**System.out.println("The shortest distances between every pair of vertices are:");**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**if (dist[i][j] == Integer.MAX\_VALUE) {**

**System.out.print("INF ");**

**} else {**

**System.out.print(dist[i][j] + " ");**

**}**

**}**

**System.out.println();**

**}**

**}**

**// Driver method**

**public static void main(String[] args) {**

**// Create a graph represented by a 2D array**

**// graph[i][j] represents the weight of the edge from vertex i to vertex j**

**int[][] graph = {**

**{0, 3, 0, 0, 0, 0},**

**{3, 0, 1, 0, 0, 0},**

**{0, 1, 0, 7, 0, 2},**

**{0, 0, 7, 0, 1, 3},**

**{0, 0, 0, 1, 0, 6},**

**{0, 0, 2, 3, 6, 0}**

**};**

**// Call the Floyd-Warshall algorithm**

**floydWarshall(graph);**

**}**

**}**

**output**

**The shortest distances between every pair of vertices are:**

**0 3 4 9 10 6**

**3 0 1 6 7 3**

**4 1 0 5 6 2**

**9 6 5 0 1 3**

**10 7 6 1 0 4**

**6 3 2 3 4 0**

**Conclusion:** Comment on the implementation of All Pair Shortest Path: Floyd Warshall Algorithm.

**Floyd-Warshall Algorithm** is one of the most fundamental algorithms used to find the shortest paths between all pairs of vertices in a weighted, directed graph. It is an example of a **dynamic programming** algorithm and is capable of handling both **positive and negative weights** in the graph, as long as there are **no negative-weight cycles**.

| Experiment No. 9 |
| --- |
| To implement N Queen problem using Backtracking method. |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 9**

**Title:** N-Queen

**Aim:** To study and implement N-Queen

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

**Theory:**

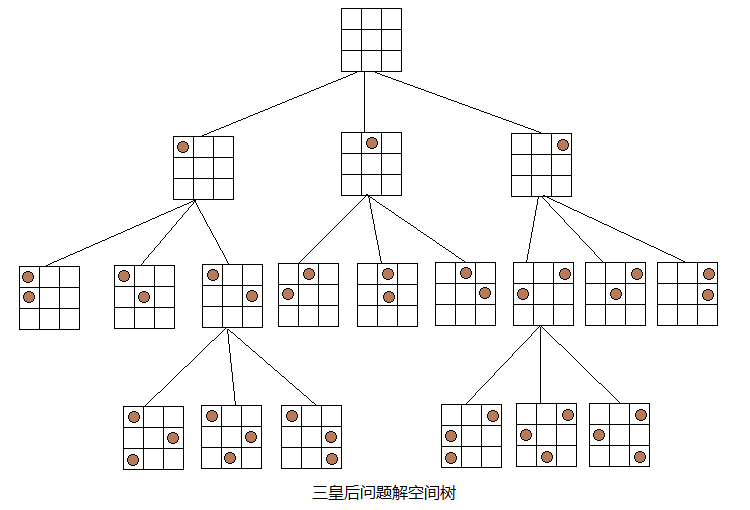
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

A black and white checkered board

AI-generated content may be incorrect.

* Start in the leftmost column
* If all queens are placed return true
* Try all rows in the current column. Do the following for every row.
  + If the queen can be placed safely in this row
    - Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    - If placing the queen in [row, column] leads to a solution then return true.
    - If placing queen doesn’t lead to a solution then unmark this [row, column] then backtrack and try other rows.
  + If all rows have been tried and valid solution is not found return false to trigger backtracking.

**Example:**



**Implementation:**

**Conclusion:** Comment on implementation of N-Queen problem.

| Experiment No. 10 |
| --- |
| 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

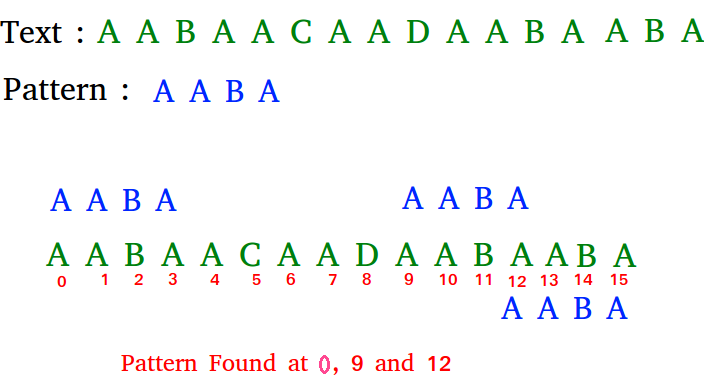
**Objective:** To introduce String matching methods

**Theory:**

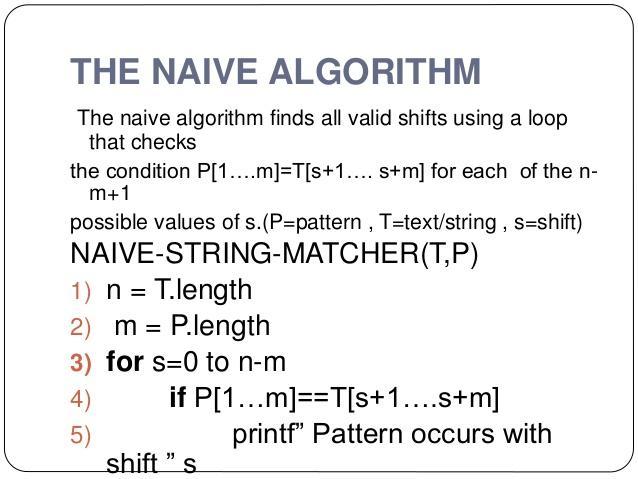
The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

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.