Raj Kumar Goel Institute of Technology, Ghaziabad

LABORATORY MANUAL

Faculty Name	:	Ms. Neha	**Department**	:	CSE
Course Name	:	Data And Analysis of Algorithms Lab	**Course Code**	:	RCS-552
Year/Sem	:	3rd/5th	**NBA Code**	:	C307
Email ID	:	nehasfcs@rkgit.edu.in	**Academic Year**	:	2018-19

Department of Computer Science and Engineering

Department of Computer Science & Engineering

VISION OF THE INSTITUTE

To continually develop excellent professionals capable of providing sustainable solutions to challenging problems in their fields and prove responsible global citizens.

MISSION OF THE INSTITUTE

We wish to serve the nation by becoming a reputed deemed university for providing value based professional education.

VISION OF THE DEPARTMENT

To be recognized globally for delivering high quality education in the ever changing field of computer science & engineering, both of value & relevance to the communities we serve.

MISSION OF THE DEPARTMENT

1. To provide quality education in both the theoretical and applied foundations of Computer Science and train students to effectively apply this education to solve real world problems.
2. To amplify their potential for lifelong high quality careers and give them a competitive advantage in the challenging global work environment.

PROGRAM EDUCATIONAL OUTCOMES (PEOs)

PEO 1: Learning: Our graduates to be competent with sound knowledge in field of Computer Science & Engineering.

PEO 2: Employable: To develop the ability among students to synthesize data and technical concepts for application to software product design for successful careers that meet the needs of Indian and multinational companies.

PEO 3: Innovative: To develop research oriented analytical ability among students to prepare them for making technical contribution to the society.

PEO 4: Entrepreneur / Contribution: To develop excellent leadership quality among students which they can use at different levels according to their experience and contribute for progress and development in the society.

.

Department of Computer Science & Engineering

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 modelling 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 team work: 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.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: The ability to use standard practices and suitable programming environment to develop software solutions.

PSO2: The ability to employ latest computer languages and platforms in creating innovative career opportunities.

COURSE OUTCOMES (COs)

C307.1	Define the basic concepts of algorithms and analyze the performance of algorithms.
C307.2	Implement various searching, sorting algorithms.
C307.3	Use various techniques for efficient algorithm design (divide-and-conquer, greedy, and dynamic algorithms) and be able to apply them while designing algorithms.
C307.4	Implement various data structures (Red-Black Tree, B-tree, Binomial Heap, Fibonacci Heap) to support specific applications.
C307.5	Know various advanced design and analysis techniques such as greedy algorithms, dynamic programming & Know the concepts of tractable and intractable problems and the classes P, NP and NP-complete problems.

CO-PO MAPPING

COs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C307.1	3	2	1	2		1						1
C307.2	2	1	1	1		1						1
C307.3	3	1	1	2		1						1
C307.4	2	1	1	1		1						1
C307.5	1	1	1	1		1						1
C307	2.2	1.2	1	1.4		1						1

CO-PSO MAPPING

	PSO1	PSO2
C307.1	1	
C307.2	1	1
C307.3	1	
C307.4	1	1
C307.5		

	1	1
C307	1	0.6

Department of Computer Science & Engineering

LIST OF EXPERIMENTS

Sr. No.	Title of experiment	Corresponding CO
1	To implement the following sorting algorithms and analyze their running time. i. Selection Sort ii. Insertion Sort iii. Bubble Sort	**C307.1**
2	To implement the following sorting algorithms. i. Merge Sort ii. Recursive Insertion Sort iii. Heap Sort iv. Quick Sort	**C307.2**
3	To implement and compare the linear search and binary search algorithms.	**C307.2**
4	To implement Count Sort linear time algorithm and analyze its running time.	**C307.2**
5	To implement Binary Search Tree operations and analyze their running time.	**C307.4**
6	To implement Insertion and Deletion operations of a Red Black Tree.	**C307.4**
7	To implement algorithms designed using the greedy programming approach to solve the following problems i. Activity Selection Problem ii. Task Scheduling Problem iii. Fractional Knapsack Problem	**C307.3**
8	To implement algorithms using Backtracking programming approach to solve the following problems i. 8-queens problem ii. Sum of Subsets Problem	**C307.5**
9	To implement following algorithms to obtain minimum spanning tree i. Kruskal's Algorithm ii. Prims Algorithm	**C307.3**

10	Implement Travelling Salesman Problem.	**C307.3**

Department of Computer Science & Engineering

Content Beyond Syllabus		
11	Implement following algorithms for single source shortest path i. Dijkstra's Algorithm ii. Bellman Ford Algorithm	**C307.3**
12	To implement algorithms designed using dynamic programming approach to solve the following problems i. Matrix Chain Multiplication ii. Problem iii. LCS Problem iv. 0-1 Knapsack Problem	**C307.3**

Department of Computer Science & Engineering

INTRODUCTION

An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi, An algorithm is a set of rules for carrying out calculation either by hand or on a machine.

1. Algorithmic is a branch of computer science that consists of designing and analyzing computer algorithms The "design" pertain to

i. The description of algorithm at an abstract level by means of a pseudo language, and
ii. ii .Proof of correctness that is, the algorithm solves the given problem in all cases.

2. The "analysis" deals with performance evaluation (complexity analysis).

The complexity of an algorithm is a function g(n) that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

There are two interpretations of upper bound.

Worst-case Complexity: The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.

Average-case Complexity: The running time for any given size input will be the average number of operations over all problem instances for a given size.

An algorithm has to solve a problem. An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

We need some way to express the sequence of steps comprising an algorithm. In order of increasing precision, we have English, pseudocode, and real programming languages. Unfortunately, ease of expression moves in the reverse order. In the manual to describe the ideasof an algorithm pseudocodes, algorithms and functios are used. In the algorithm analysis and design lab various stratgies such as Divide and conquer techinque , greedy technique and dynamic programming techniques are done. Many sorting algorithms are implemented to analyze the time complexities.String matching algorithms, graphs and spanning tree algorithms are implemented so as to able to understand the applications of various design stratgies.

Department of Computer Science & Engineering

PREFACE

Most of the professional programmers that I've encountered are not well prepared to tackle algorithm design problems. This is a pity, because the techniques of algorithm design form one of the core practical *technologies* of computer science. Designing correct, efficient, and implementable algorithms for real-world problems is a tricky business, because the successful algorithm designer needs access to two distinct bodies of knowledge:

- ***Techniques*** - Good algorithm designers understand several fundamental algorithm design techniques, including data structures, dynamic programming, depth-first search, backtracking, and heuristics. Perhaps the single most important design technique is *modeling* , the art of abstracting a messy real-world application into a clean problem suitable for algorithmic attack.
- ***Resources*** - Good algorithm designers stand on the shoulders of giants. Rather than laboring from scratch to produce a new algorithm for every task, they know how to find out what is known about a particular problem. Rather than reimplementing popular algorithms from scratch, they know where to seek existing implementations to serve as a starting point. They are familiar with a large set of basic algorithmic problems, which provides sufficient source material to model most any application.

This manual on algorithm design, providing access to both aspects of combinatorial algorithms technology for computer professionals and students. Though all the efforts have been made to make this manual error free, yet some errors might have crept in inadvertently. Suggestions from the readers for the improvement of the manual are most welcomed.

Ms. Neha,
Ms. Preeti Gupta
(Assistant Professor, Dept. of CSE)

Department of Computer Science & Engineering

DO'S AND DONT'S

DO's

1. Conform to the academic discipline of the department.
2. Enter your credentials in the laboratory attendance register.
3. Read and understand how to carry out an activity thoroughly before coming to the laboratory.
4. Ensure the uniqueness with respect to the methodology adopted for carrying out the experiments.
5. Shut down the machine once you are done using it.

DONT'S

1. Eatables are not allowed in the laboratory.
2. Usage of mobile phones is strictly prohibited.
3. Do not open the system unit casing.
4. Do not remove anything from the computer laboratory without permission.
5. Do not touch, connect or disconnect any plug or cable without your faculty/laboratory technician's permission.

Department of Computer Science & Engineering

GENERAL SAFETY INSTRUCTIONS

1. Know the location of the fire extinguisher and the first aid box and how to use them in case of an emergency.

2. Report fires or accidents to your faculty /laboratory technician immediately.

3. Report any broken plugs or exposed electrical wires to your faculty/laboratory technician immediately.

4. Do not plug in external devices without scanning them for computer viruses.

Department of Computer Science & Engineering

GUIDELINES FOR LABORTORY RECORD PREPARATION

While preparing the lab records, the student is required to adhere to the following guidelines:

Contents to be included in Lab Records:

1. Cover page
2. Vision
3. Mission
4. PEOs
5. POs
6. PSOs
7. COs
8. CO-PO-PSO mapping
9. Index
10. Experiments

    Aim

    Source code

    Input-Output

A separate copy needs to be maintained for pre-lab written work

The student is required to make the Lab File as per the format given on the next two pages.

Department of Computer Science & Engineering

Raj Kumar Goel Institute of Technology, Ghaziabad

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DESIGN AND ANALYSIS OF ALGORITHMS LAB (RCS-552)

Name	
Roll No.	
Section- Batch	

Department of Computer Science & Engineering

INDEX

Experiment No.	Experiment Name	Date of Conduction	Date of Submission	Faculty Signature

Department of Computer Science & Engineering

GUIDELINES FOR ASSESSMENT

Students are provided with the details of the experiment (Aim, pre-experimental questions, procedure etc.) to be conducted in next lab and are expected to come prepared for each lab class.

Faculty ensures that students have completed the required pre-experiment questions and they complete the in-lab programming assignment(s) before the end of class. Given that the lab programs are meant to be formative in nature, students can ask faculty for help before and during the lab class.

Students' performance will be assessed in each lab based on the following Lab Assessment Components:

AC1: Written Work (Max. marks = 5)
AC2: Fundamental Knowledge to conduct Experiment (Max. marks = 5)
AC3: Experiment Completed Successfully (Max. marks = 5)
AC4: Questions Answered (Max. marks = 5)
AC5: Punctuality (Max. marks = 5)

In each lab class, students will be awarded marks out of 5 under each component head, making it total out of 25 marks.

Department of Computer Science & Engineering

EXPERIMENT – 1

Aim: To implement the following sorting algorithms and analyze their running time.

i. Selection Sort
ii. Insertion Sort
iii. Bubble Sort

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

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.

Pre-experiment Questions:

Q1. What is an algorithm?

Q2. What do you mean by analysis of an algorithm?

Q3.What do you understand by asymptotic notation?

Q4.How many asymptotic notations are you aware of? Name them.

Q5.What is theta notation?

Q6.What do you understand by upper and lower bounds?

Q7.What are tight and loose bounds?

Q8.With the help of an example explain how selection sort works. Q9.With the help of an example explain how insertion sort works. Q10.With the help of an example explain how bubble sort works.

Pseudo-code:

SELECTION-SORT*(A)*

n ← length[A]

for j ← 1 **to** n −

1 **do** smallest

← j **for** i ← j +

1 **to** n

do if A[i] < A[smallest]

then smallest ← i

exchange A[j] ↔

A[smallest]

```
INSERTION-SORT(A)

    for j ← 2 ton c1 n

        do key ← A[ j ]

            i←j−1

                while i > 0 and A[i ] >
                        key
```

do A[i + 1] ← A[i
]

i ←i −
1

A[i + 1] ← key

BUBBLE-SORT(A)

do

swapped := false

for each i **in** 0 **to** length(A) - 2 **do:**

if A[i] > A[i + 1] **then**

swap(A[i], A[i + 1])

swapped :=
true

end

if end for

while swapped

Input:

For each sorting algorithm the input should be an array of n elements

Output:

For each sorting algorithm the output should be the elements of the sorted array after each iteration.

Post –Experiment Questions:

Q1. What is the worst case, average case and best case running time of Selection Sort? How do you get it?

Q2. What is the worst case, average case and best case running time of Insertion Sort?

How do you get it?

Q3. What is the worst case, average case and best case running time of Bubble Sort? How do you get it?

Q4. What is the role of variable swapped in Bubble Sort algorithm? 5) Compare the three algorithms that you have implemented.

Department of Computer Science & Engineering

EXPERIMENT - 2

Aim: To implement the concept of divide and conquer strategy of designing algorithms for the following sorting algorithms and analyze their running time.

i. Merge Sort
ii. Recursive Insertion Sort
iii. Heap Sort
iv. Quick Sort

Description:

Merge sort is an O(n log n) comparison-based sorting algorithm. Conceptually, a merge sort works as follows:

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

Pre-Experiment Questions:

Q1. What is the divide and conquer strategy of designing algorithms?

Q2.How are these algorithms analyzed?

Q3.How are recurrence equations written for divide and conquer algorithms? Q4.What are the various methods of solving recurrences?

Q5.Describe the substitution method for solving recurrences. Describe the iterative method for solving recurrences.

Q6.Describe the recursion tree method for solving recurrences. Q7.What is the Master's Theorem?

Q8.Can Master's theorem be used to solve all recurrences?

Q9.When can Master's method not be used for solving recurrences?

Pseudo-code:

```
MERGE-SORT(A, p, r )
        if p < r
                then q = FLOOR((p +
        r)/2) MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r
        ) MERGE(A, p, q, r )
```

Initial call: MERGE-SORT(A, 1, n) MERGE(A, p, q, r)

```
        n1 ← q − p +

        1 n2 ←r − q

        create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
        for i ← 1 to n1
                do L[i ] ← A[p + i − 1]
        for j ← 1 to n2
```

```
        do R[ j ] ← A[q +

        j ] L[n1 + 1] ← ∞

        R[n2 + 1] ← ∞

        i ← 1 j ← 1

        for k ← p to r

        do if L[i ] ≤ R[ j ]

                then A[k] ← L[i ]

                        i ← i + 1

                else A[k] ← R[ j ]

        j ← j+1

RECURSIVE-INSERTION-SORT(A,
                n)
    if n ≠ 1
    then RECURSIVE-INSERTION-SORT(A,n-1)

        key ← A[ n ]

        i ← n−1

        while i > 0 and A[i ] >

                key do A[i + 1] ←

                A[i ] i ← i − 1

        A[i + 1] ← key

Heapify(A, i)
{
    l = Left(i); r = Right(i);
    if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
                                else
```

```
                                        largest = i;
    if (r <= heap_size(A) && A[r] > A[largest])
            largest = r;
    if (largest != i)
            Swap(A, i,
            largest);
            Heapify(A,
            largest);
}
```

```
BuildHeap(A)
{
        heap_size(A) = length(A);
        for (i = floor(length[A]/2) downto 1)
                Heapify(A, i);
}
Heapsort(A)
{
                                                BuildHeap(A);
                                                for (i = length(A) downto 2)
                                                {
                        Swap(A[1], A[i]); heap_size(A) -=
                        1; Heapify(A, 1);
                }
}
QUICKSORT(A, p, r )
        if p < r
        then q ←PARTITION(A, p, r )

                QUICKSORT(A, p, q −

                1) QUICKSORT(A, q +
                1, r )
```

Initial call is QUICKSORT(A, 1, n).

```
        PARTITION(A, p, r )
                x ← A[r ] i←p−1
                for j ← p to r − 1
                        do if A[ j ] ≤ x then i
                        ← i + 1

return i + 1
```

exchange A[i] ↔ A[j]

exchange A[i + 1] ↔ A[r]

Department of Computer Science & Engineering

Input:

Array of n elements.

Output:

Sorted array with intermediate results as follows

Merge Sort:- After each Merge Operation.

Recursive Insertion Sort:- Each time an element is inserted.

Heap Sort:- Array after BuildHeap() is called and after each iteration of the main loop in HeapSort().

Quick Sort:- After each partition operation.

Post-Experiment Questions:

Q1.What are the worst case, best case and average case performances of each of the algorithms implemented?

Q2.How will you analyze Merge Sort?

Q3.What is the recurrence of Recursive Insertion

Sort? Q4.What is a heap?

Q5.How can you use heap to sort in descending

order? Q6.What is pivot in QuickSort?

Q7.How does the pivot affect the performance of QuickSort?

Q8.Can you use Quick Sort to sort in descending order?

How? Q9.What is randomized QuickSort?

Q10. Do you think that Merge Sort has some drawback? If yes, what is the drawback?

Q11. Compare Heap Sort and Quick Sort.

Department of Computer Science & Engineering

EXPERIMENT - 3

Aim: To implement and compare the linear search and binary search algorithms.

Description:

Linear search or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. The list need not be ordered. Linear search is the simplest search algorithm; it is a special case of brute-force search.

Binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomic divide-and-conquer search algorithm and executes in logarithmic time. The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Pre-Experiment Questions:

Q1.Why is searching operation important in computer

science? Q2.How do the linear search and binary search work?

Q3.What is the worst case running time of linear search? What is the worst case running time of binary search?

Q4. What is the pre requisite for using binary search?

Department of Computer Science & Engineering

Pseudo- code:

```
LINEAR-SEARCH(A, key)
        for i = 1 to length(A)
            if a[ i ] = key
            then return
        i return 0

BINARY-SEARCH(A,
        key) l=1 ; r =
        length(A) while l <
        r
                do mid = floor((l + r)/2)
                if key < A[mid]
                        then r = mid -1
                else if key > A[mid]
                        then l = mid + 1
                else return mid
        return 0
```

Input:

Array of n elements.

Output:

The output is the index of the key if it is found and a message if key is not found. In case of binary search if the array is not sorted then a message is displayed indicating that binary search cannot be used on unsorted array.

Post-Experiment Questions:

Q1. Prove that binary search running time of binary search is lg

n. Q2. What is the drawback of binary search?

Department of Computer Science & Engineering

EXPERIMENT - 4

Aim: To implement Count Sort linear time algorithm and analyze its running time.

Description:

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

The algorithm loops over the items, computing a histogram of the number of times each key occurs within the input collection. It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the items again, moving each item into its sorted position in the output array.

Pre-Experiment Questions:

Q1.What are comparison sort algorithms?

Q2. What is the best performance that comparison sort algorithms can

give? Q3. What are decision trees?

Q4. Name some algorithms that can give linear performance

Q5. What are the limitations of Count Sort?

Q6. What are the limitations of Radix Sort?

Q7. What are the limitations of Bucket

Sort?

Q8. What is the worst case performance of Bucket Sort? When does this happen?

Department of Computer Science & Engineering

Pseudo- code:

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] =
A[j]; C[A[j]] -= 1;
```

Input:

Array of n elements.

Output :

Arrays B and C should be shown after each loop. Finally, the sorted array should be displayed.

Post-Experiment Questions:

Q1. Analyze Count Sort.

Q2. Give some applications of Count Sort

Department of Computer Science & Engineering

EXPERIMENT - 5

Aim: To implement Binary Search Tree operations and analyze their running time.

Description:

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree. (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.)

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

Pre-Experiment Questions:

Q1.What is a BST?

Q2.What is the application of BST?

Q3.Why is it not advisable to use arrays for deriving the benefits of binary search?

Q4.How does the search operation work on a BST?

Q5.How is a node inserted in a BST?

Q6. What are the three cases of deletion? How are they handled?

Pseudo-code:

Root[T] points to the root of the tree
Each Node contains the following fields:-

key (and possibly other satellite data).
left: points to left child.
right: points to right child.
p: points to parent. *p*[*root*[*T*]] = NIL.

TREE-INSERT*(T, z)*

y ← NIL

x ← *root*[*T*]

while *x* ≠ NIL **do**

y ← *x*

if *key*[*z*] < *key*[*x*]

then *x* ← *left*[*x*]

else *x* ← *right*[*x*]

p[*z*] ← *y*

if *y* = NIL

then *root*[*T*] ← *z*

else if *key*[*z*] < *key*[*y*]

then *left*[*y*] ← *z* **else** *right*[*y*] ← *z*

TREE-DELETE*(T, z)*

// Determine which node *y* to splice out: either *z* or *z*.s successor.

if *left*[*z*] = NIL or *right*[*z*] = NIL

then *y* ← *z*

else *y* ← TREE-SUCCESSOR*(z)*

// *x* is set to a non-NIL child of *y*, or to NIL if *y* has no children.

if *left*[*y*] ≠ NIL **then** *x* ← *left*[*y*]

else $x \leftarrow right[y]$

// y is removed from the tree by manipulating pointers of $p[y]$ and x.

if $x \neq$ NIL

then $p[x] \leftarrow p[y]$

if $p[y]$ = NIL

then $root[T] \leftarrow x$

```
        else    if y = left[p[y]]
                then left[p[y]] ← x
                else right[p[y]] ←
                x
        // If it was z.s successor that was spliced out, copy its data into z.
        if y ≠ z
        then key[z] ← key[y]
                copy y.s satellite data into z
        return y

TREE-SUCCESSOR(x
        )
        if right[x] ≠ NIL
        then return TREE-MINIMUM(right[x])
        y ← p[x]
        while y ≠ NIL and x = right[y]
        do      x←y
                y ← p[y]
        return y

TREE-MINIMUM(x
        )
        while left[x] ≠ NIL
        do x ← left[x]
        return x

INORDER-TREE-WALK(x)
        if x _= NIL
        then INORDER-TREE-WALK(left[x])
                print key[x]
                INORDER-TREE-WALK(right[x])
```

```
TREE-SEARCH(x, k)
    if x = NIL or k = key[x]
    then return x
```

if $k < key[x]$
then return TREE-SEARCH*(left*[*x*], *k)*
else return TREE-SEARCH*(right*[*x*], *k)*

Initial call is TREE-SEARCH*(root*[*T*], *k)*.

Input:
Set of n elements
Output:

In-order traversal after each operation on the tree.

Post-Experiment Questions:

Q1. What is the time taken by the various operations of a BST? Justify your answer

Q2. Suppose the elements are inserted in ascending order of the key, what will the BST look like and how much running time will the search operation require?

Department of Computer Science & Engineering

EXPERIMENT - 6

Aim: To implement Insertion and Deletion operations of a Red Black Tree.

Description:

A red–black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as text fragments or numbers.

The leaf nodes of red–black trees do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

1. A node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. The uniform number of black nodes in the paths from root to leaves is called the black-height of the red–black tree.

These constraints enforce a critical property of red–black trees: the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst case, unlike ordinary binary search trees.

Department of Computer Science & Engineering

Pre-Experiment questions

Q1.What is a Red Black Tree?

Q2.What is the need of an

RBT?

Q3.Is there any other data structure that can be used for meeting the need?

Q4.Can I have a GREEN BLUE Tree instead of RED BLACK Tree?

Q5.Which property of RBT may be disturbed if a node is inserted or

deleted? Q6.What are rotations?

Q7.What are the various cases of inserting a node in an RBT? How are they handled?

Q8.What are the various cases of deleting a node from an RBT? How are they

handled?

Pseudo- code:

The node structure is same as that for a BST except that an additional field *color* is added.

LEFT-ROTATE*(T, x)*

$y \leftarrow right[x]$

$right[x] \leftarrow left[y]$

if $left[y] \neq nil[T]$

then $p[left[y]] \leftarrow$

$x\ p[y] \leftarrow p[x]$

if $p[x] = nil[T]$

then $root[T] \leftarrow y$

else if $x = left[p[x]]$

then *left*[*p*[*x*]] ← *y* **else**

right[*p*[*x*]] ← *y*

left[*y*] ←

x p[*x*] ←

y

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

RB-INSERT*(T, z)*

```
y ← nil[T ]
x ← root[T ]
while x ≠ nil[T
] do      y←x
        if key[z] < key[x]
        then x ← left[x] else x
        ← right[x]
p[z] ← y
if y = nil[T ]
then root[T ] ← z
else    if key[z] < key[y]
then left[y] ← z else right[y]
← z left[z] ← nil[T ]
right[z] ← nil[T ]
color[z] ← RED
```

RB-INSERT-FIXUP*(T, z)*

RB-INSERT-FIXUP*(T, z)*

```
while color[p[z]] = RED
        if p[z] = left[p[p[z]]]
                then y ← right[p[p[z]]]
                if color[y] = RED
                        then color[p[z]] ←
```

BLACK $color[y] \leftarrow$ BLACK

$color[p[p[z]]] \leftarrow$ RED

$z \leftarrow p[p[z]]$

else if $z = right[p[z]]$

then $z \leftarrow p[z]$

LEFT-ROTATE*(T, z)*

color[*p*[*z*]] ← BLACK *color*[*p*[*p*[*z*]]] ← RED

RIGHT-ROTATE*(T, p*[*p*[*z*]]*)*

else (same as **then** clause with right and left exchanged)

color[*root*[*T*]] ← BLACK

RB-DELETE*(T, z)*

if *left*[*z*] = *nil*[*T*] or *right*[*z*] = *nil*[*T*]

then *y* ← *z*

else *y* ← TREE-SUCCESSOR*(z)*

if *left*[*y*] ≠ *nil*[*T*]

then *x* ← *left*[*y*] **else** *x*

← *right*[*y*]

p[*x*] ← *p*[*y*]

if *p*[*y*] = *nil*[*T*]

then *root*[*T*] ← *x*

else **if** *y* = *left*[*p*[*y*]]

then *left*[*p*[*y*]] ←

x

else *right*[*p*[*y*]] ←

x

if *y* ≠ *z*

then *key*[*z*] ← *key*[*y*]

copy *y*.s satellite data into *z*

if *color*[*y*] = BLACK

then RB-DELETE-FIXUP*(T, x)*

return *y*

RB-DELETE-FIXUP*(T, x)*

```
while x ≠ root[T] and color[x] = BLACK
    if x = left[p[x]]
    then w ← right[p[x]]
        if color[w] = RED
```

then *color*[*w*] ← BLACK

color[*p*[*x*]] ← RED

LEFT-ROTATE*(T, p*[*x*]*)*

w ← *right*[*p*[*x*]]

if *color*[*left*[*w*]] = BLACK and *color*[*right*[*w*]] = BLACK

then *color*[*w*] ← RED

x ← *p*[*x*]

else if *color*[*right*[*w*]] = BLACK

then *color*[*left*[*w*]] ← BLACK

color[*w*] ← RED

RIGHT-ROTATE*(T,w)*

w ← *right*[*p*[*x*]]

color[*w*] ← *color*[*p*[*x*]]

color[*p*[*x*]] ← BLACK

color[*right*[*w*]] ← BLACK

LEFT-ROTATE*(T, p*[*x*]*)*

x ← *root*[*T*]

else (same as **then** clause with right and left exchanged)

color[*x*] ← BLACK

Input:
Set of n elements.

Output:
In-order traversal after each operation on RB tree.

Post-Experiment Questions:

Q1.What is running time of RB-INSERT operation?

Q2. What is the running time of RB-DELETE operation?

Department of Computer Science & Engineering

EXPERIMENT - 7

Aim: To implement algorithms designed using the greedy programming approach to solve the following problems

iv. **Activity Selection Problem**
v. **Task Scheduling Problem**
vi. **Fractional Knapsack Problem**

Description:

The activity selection problem is a combinatorial optimization problem concerning the selection of non-conflicting activities to perform within a given time frame, given a set of activities each marked by a start time (si) and finish time (fi). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming that a person can only work on a single activity at a time.

Task scheduling problem is the problem of scheduling unit-time tasks with deadlines and penalities for a single processor has following inputs:

S={a1,a2,...,an} of n unit-time task. A unit-time task requires exactly 1 unit of time to complete

deadlines d1,d2,...,dn, 1<=di<=n

penalities w1,w2,...,wn. A penality wi is incurred if task ai is not finished by time di, and no penality if task finishes at deadline

The problem is to find a schedule for S that minimizes the total penalty incurred for missed deadlines

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight 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. As 0-1 knapsack, here we can take fractions of an item.

Department of Computer Science & Engineering

Pre-Experiment Questions:

Q1.What do you understand by Greedy Programming Approach of Solving Problems? Q2.How does this approach differ from Dynamic Approach?

Q3.What are the elements of Greedy Programming? Q4.What is the Greedy Choice Property?

Q5.Is there any advantage of using Greedy approach over Dynamic Approach?

Q6.Can you use Greedy Approach in all problems which can be solved using Dynamic Approach? What about vice versa?

Q7.How does fractional knapsack problem differ from 0-1 knapsack problem? Q8.What is Activity Selection Problem?

Q9.How will you solve it using Greedy Approach? Explain Task Scheduling Problem. Q10. What do you understand by Canonical Form of a schedule?

Q11. How will you solve it using Greedy Approach?

Q12. Name some other problems that are ideal for solving using Greedy Approach. Q13. Does Greedy Approach always give optimal solution?

Pseudo- code:

GREEDY-ACTIVITY-SELECTOR*(s, f, n)*

$A \leftarrow \{a_1\}$ $i \leftarrow 1$ **for** m $\leftarrow 2$ **to** n

do **if** $s_m \geq f_i$

then $A \leftarrow A \cup \{a_m\}$

$i \leftarrow m$

return A

Department of Computer Science & Engineering

```
GREEDY-JOB( d, T, n)
        d[0] = T[0] = 0
        T[1] =
        1 k=1
        for i = 2 to n do

        r=k
                while d[T[r]] > d[I] AND d[T[r]] ≠ r
                do      r=r-1
                        if d[T[r]] <= d[i] AND d[i] > r
                        then
                                for q = r+1 downto k
                                do      T[q+1] = T[q]
```

Sort the tasks is descending order of penalties before executing the algorithm

FRACTIONAL-KNAPSACK(v,w,W)

$load \leftarrow 0$

$i \leftarrow 1$

while $load < W$ and $i \leq n$

if $w_i \leq W - load$

then take all of item i

else take $(W - load)/w_i$ of item i

add what was taken to

load

$i \leftarrow i + 1$

Department of Computer Science & Engineering

Input:

Activity Selection Problem: The list of activities with their start and finish time.

Task Scheduling Problem: The list of tasks with their start and finish time.

Fractional knapsack problem: The index of the item along with the quantity (1, 0 or a fraction)

Output:

Activity Selection Problem: The list of activities scheduled in optimal manner.

Task Scheduling Problem: Task schedule in canonical form

Fractional knapsack problem: The index of the item along with the quantity (1, 0 or a fraction)

Post-Experiment Questions:

Q1.What is the complexity of each of the algorithms implemented? Q2.give a practical application of each of the problems

Department of Computer Science & Engineering

EXPERIMENT - 8

Aim: To implement algorithms using Backtracking programming approach to solve the following problems

iii. **8-queens problem**
iv. **Sum of Subsets Problem**

Descritption:

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The problem can be quite computationally expensive as there are 4,426,165,368 (i.e., 64C8) possible arrangements of eight queens on an 8×8 board, but only 92 solutions. It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute-force computational techniques. For example, just by applying a simple rule that constrains each queen to a single column (or row), though still considered brute force, it is possible to reduce the number of possibilities to just 16,777,216 (that is, 88) possible combinations. Generating permutations further reduces the possibilities to just 40,320 (that is, 8!), which are then checked for diagonal attacks.

Sum of subset problem: Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Pre-Experiment Questions:

Q1.What do you understand by

Backtracking? Q2.What is 8-queens

problem?

Q3.Explain Sum of Subset problem?

Q4.How the backtracking is applied in sum of subset problem?

Q5.Name some other problems that are ideal for solving using Backtracking Approach

Department of Computer Science & Engineering

Pseudo- code:

```
Algorithm Place (k, i)
// Return TRUE if a queen can be placed in k^th row and ith column. Otherwise it returns
// FALSE. X[] is a global array whose first (k-1) values have been set. Abs(r ) returns the
// absolute value of r.
{
   for j := 1 to k-1 do
         if ((x[j] =i) or (Abs(x[j]-i)= Abs(j-k)))
               then return FALSE;
   return TRUE;
}

Algorithm NQueens(k, n)

// Using backtracking, this procedure prints all possible placements of n queens on an //
n X n chessboard so that they are nonattacking.
 {
     for i := 1 to n do
        {
            if Place(k, i) then
            {
                 x[k] := i;
                 if (k = n) then write (x[1 : n]);
else NQueens(k+1, n);

            }
        }
}

Algorithm SumOfSub(s, k, r)

// Find all subsets of w[1: n] that sum to m. The values of x[j], 1 belongs j < k, have

already been determined. S = <= w[j] * x[j] and r = <= w[j]. The w[j]'s are in

non-decreasing order.
```

```
{
  // Generate left child. Note s +w[k]  □ m since B_{k-1} is true
  x[k] :=1;
  if (s + w[k] = m) then write (x[1 : k]);
else if (s + w[k] + w[k+1] <= m)
        then SumOfSub(s + w[k], k+1, r- w[k]);
  if (s + r -w[k] <= m) and (s + w[k+1] <= m)) then
  {
     x[k]:=0;
     SumOfSub(s, k+1, r-w[k]);
  }
}
```

Input:

8-Queens Problem: Solution set that represents the position of the queen to be placed. Initialized to *nil*.

Sum of Subset: Weight of a set of *n* elements (w1, w2, w3….wn) and the total expected sum *m*.

Output:

8-Queens Problem: Solution set that represents the position of the queen to be placed

Sum of Subset: All solution subsets where each solution subset is represented by an n- tuple (x1, x2, x3….xn).

Post -Experiment Questions:

Q1.What is the complexity of each of the algorithms implemented

Q2.Draw state space tree for sum of subset problem .

Q3.Find solution for 4-Queens problem using the same algorithm.

Q4.Can you find solution for any no. of Queens using Algorithm for N-Queens

Department of Computer Science & Engineering

EXPERIMENT - 9

Aim: To implement following algorithms to obtain minimum spanning tree

iii. **Kruskal's Algorithm**
iv. **Prims Algorithm**

Description:

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is agreedy algorithm in graph theory as it finds a minimum spanning tree for aconnected weighted graph adding increasing cost arcs at each step. 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. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

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.

Pre-Experiment Questions:

Q1.What do you understand by Greedy Programming Approach of Solving

Problems? Q2. How Minimum Spanning Tree uses Greedy Approach

Q3.Explain the working of Kruskal's

Algorithm Q4.What are the various

applications of MST? Q5.Can we use dynamic

approach in MST?

Q6.How does the working of Kruskal's algorithm differ from Prim's Algorithm

Q7.What is Prim's Algorithm

Q8.How Greedy Approach is applicable in Prim's Algo.?

Q9.Which of these algorithms gives an optimal solution?

Q10. Which among these algorithms gives a forest and

why?

Pseudo-code:

Kruskal's algorithm:

MST-KRUSKAL(G,

w) A=nil

For each vertex v ∈ V[G]

do MAKE-SET(v)

sort the edges of E into nondecreasing order by weight w

for each edge (u,v) ∈ E, taken in nondecreasing order by

weight do if FIND-SET(u) ≠ FIND-SET(v)

then A ,<− A U {(u,

v)} UNION (u,

v)

return A

Prim's Algorithm:

1. Associate with each vertex *v* of the graph a number *C*[*v*] (the cheapest cost of a connection to*v*) and an edge *E*[*v*] (the edge providing that cheapest connection). To initialize these values, set all values of *C*[*v*] to +∞ (or to any number larger than the

maximum edge weight) and set each $E[v]$ to a special flag value indicating that there is no edge connecting v to earlier vertices.

2. Initialize an empty forest F and a set Q of vertices that have not yet been included in F(initially, all vertices).
3. Repeat the following steps until Q is empty:

a. Find and remove a vertex v from Q having the minimum possible value of $C[v]$
b. Add v to F and, if $E[v]$ is not the special flag value, also add $E[v]$ to F
c. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has smaller weight than $C[w]$, perform the following steps:
    i. Set $C[w]$ to the cost of edge vw
    ii. Set $E[w]$ to point to edge vw.

Input:

Number of nodes in a graph and the corresponding adjacency matrix.

Output:

Kruskal's Algorithm: Minimum wt. edges are selected so as to get the MST

Prims Algorithm: Minimum wt. edges are selected from a particular source node so as to get the MST

Post-Experiment Questions:

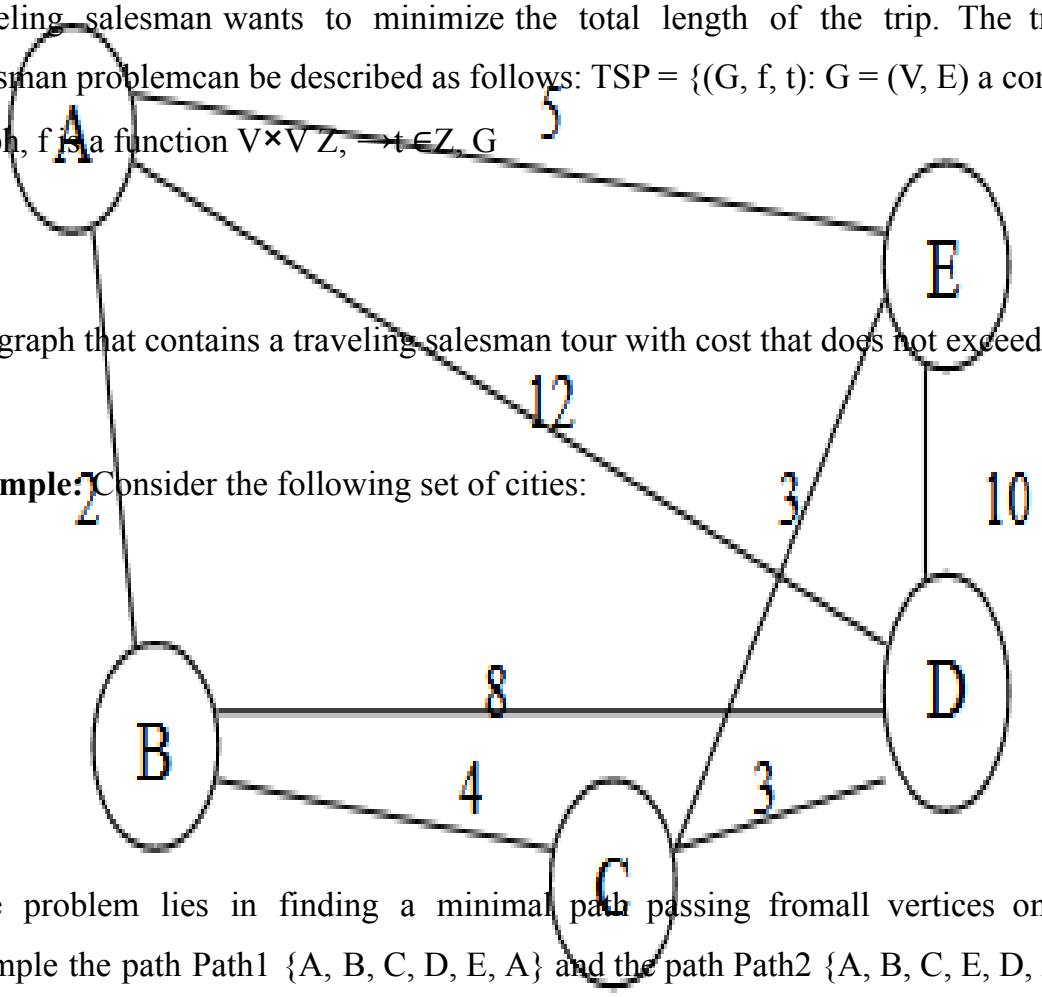
Q1.What is the complexity of each of the algorithms implemented? Q2. Give a practical application of each of the problems.

Department of Computer Science & Engineering

EXPERIMENT - 10

Aim: **Implement Travelling Salesman Problem.**

Description: The traveling salesman problemconsists of a salesman and a set of cities. The salesman has to visit each one of the cities starting froma certain one (e.g. the hometown) and returning to the same city. The challenge of the problemis that the traveling salesman wants to minimize the total length of the trip. The traveling salesman problemcan be described as follows: TSP = {(G, f, t): G = (V, E) a complete graph, f is a function V×V Z, →t ∈Z, G



is a graph that contains a traveling salesman tour with cost that does not exceed t }.

Example: Consider the following set of cities:

The problem lies in finding a minimal path passing fromall vertices once. For example the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31

Pre-Experiment questions

Q.1 What is traveling salesman problem?

Q.2 What is adjancy matrix?

Q.3 What is Hamiltonian cycle?

Q.4 How will you solve it using Dynamic Programming

Approach? Q.5 How will you solve it using Greedy Approach?

Department of Computer Science & Engineering

Pseudo- code

Approx-TSP-Tour(G = (*V, E*), c)

1. select a vertex r ∈ V[G] to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-Prim(G, c, r)
3. let L be the list of vertices visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H that visits the vertices in the order L

END

MST-PRIM (*G* = (*V,E*), *c, r*)

1. **for** each vertex $u \in V[G]$
2. **do** key[u] **:=** ∞
3. π[u] **:=** `NIL` */*π[u] ia parent of node u */*
4. key[r] := 0
5. $Q := V[G]$ */*Q is a priority queue of set of all vertices not in the tree */*
6. while $Q \neq \{\}$
7. **do** u:= EXTRACT-MIN(Q)
8. **for** each $v \in$ Adj[u]
9. **do if** $v \in Q$ and $c(u,v) <$ key[v]
10. then π[v] := u
11. key[v] := $c(u,v)$

END

Input:

Number of nodes in a graph and adjacency matrix with all the edge weights.

Output :

Shortest Hamiltonian cycle from a source node to cover all other return back to sourse node.

Department of Computer Science & Engineering

Post-Experiment Questions:

Q1. What is the Time complexity of Traveling salesman problem?

Q2. Give a practical application of Traveling salesman problem

algorithm.. Q.3 Explain Negative Weight Cycle?

Q.4 How can we solve TSP using branch and bound

Department of Computer Science & Engineering

EXPERIMENT - 11

Aim: **Implement following algorithms for single source shortest path**

i. **Dijkstra's Algorithm**
ii. **Bellman Ford Algorithm**

Description:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that those intersections have not yet been visited; some variants of this method simply leave the intersections' distances unlabeled. Now, at each iteration, select the current intersection. For the first iteration, the current intersection will be the starting point, and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be the closest unvisited intersection to the starting point (this will be easy to find).

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single

source vertex to all of the other vertices in a weighted digraph.[1] It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges. Since the longest possible path without a cycle can be |V|-1 edges, the edges must be scanned |V|-1 times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length |V| edges has been found which can only occur if at least one negative cycle exists in the graph.

Department of Computer Science & Engineering

Pre-Experiment questions

Q1.What do you understand by Single Source Shortest path? Q2.What is shortest path problem?

Q3. Explain Negative Weight Cycle?

Q4. Can you apply Dijkstra's algorithm in a graph with negative weight edges. Q5.Explain how Bellman Ford algorithm differ from Dijksra's algorithm

Q6. Can positive weight cycle lead to a solution using any one of these algorithms? Q7. What do you understand by Relaxation?

Q8.Explain the working of both the algorithms.

Q9 How can you find the shortest path with single source in directed acyclic graph?

Pseudo- code

Dijkstra's algorithm:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their*tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Bellman Ford's algorithm:

```
BellmanFord(list vertices, list edges, vertex source)
  ::distance[],predecessor[]

  // This implementation takes in a graph, represented as lists of vertices and edges, and fills two arrays (distance and predecessor) with shortest-path (less cost/distance/metric) information

  // Step 1: initialize graph
  for each vertex v in vertices:
     distance[v] := inf              // At the beginning , all vertices have a weight of infinity
     predecessor[v] := null          // And a null predecesor

   distance[source] := 0                        // Except for the Source, where the Weight is zero

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
     for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
           distance[v] := distance[u] + w
           predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
     if distance[u] + w < distance[v]:
```

```
        error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

Input:

Number of nodes in a graph and adjacency matrix with all the edge weights.

Output :

Dijkstra's Algorithm: Shortest path from a source node to all other nodes for all positive weight edges

Bellman Ford Algorithm: Shortest path from a source node to all other nodes if there is no negative weight cycle

Post-Experiment Questions:

Q1. What is the complexity of each of the algorithms

implemented? Q2. Give a practical application of each of the

algorithm.

Department of Computer Science & Engineering

EXPERIMENT - 12

Aim: To implement algorithms designed using dynamic programming approach to solve the following problems

i. **Matrix Chain Multiplication Problem**
ii. **LCS Problem**
iii. **0-1 Knapsack Problem**

Description:

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to high-level subproblems often reuse lower level subproblems. Problems with these two properties—optimal substructure and overlapping subproblems—can be approached by a problem-solving technique called dynamic programming, in which subproblem solutions are memoized rather than computed over and over. The procedure requires memoization—saving the solutions to one level of subproblem in a table (analogous to writing them to a memo, hence the name) so that the solutions are available to the next level of subproblems.

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight 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. Given a set of n items numbered from 1 up to n, each with a weight wi and a value vi, along with a maximum weight capacity W,

maximize

subject to and

Here xi represents the number of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

Pre-Experiment Questions

Q1.What do you understand by Dynamic Programming? What is the benefit of using Dynamic Programming?

Q2.What are optimization Problems?

Q3.What do you understand by overlapping subproblems? Q4.What is optimal substructure property?

Q5.What are the elements of dynamic programming? What is the Matrix Chain Multiplication Problem?

Q6.How will you solve it using Dynamic Programming Approach? Q7.What is LCS Problem?

Q8.How will you solve it using Dynamic Programming Approach? Q9.What is 0-1 Knapsack Problem?

Q10. How will you solve it using Dynamic Programming Approach?

Q11. Name some other problems that you can solve using Dynamic Programming Approach

Department of Computer Science & Engineering

Pseudocode:

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for i ← 1 to n
        m[i, i] ← 0
    for l ← 2 to n
        for i ← 1 to
            n-l+1 j ←
            i+l-1 m[i,
            j] ← ∞
            for k ← i to j-1
                q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                if q < m[i, j]
                    m[i, j] ← q s[i,
                        j]
                    ← k
    return m and s

PRINT-OPTIMAL-PARENS(s, i, j)
    if i = j
    then print "A"
    else print "("
        PRINT-OPTIMAL-PARENS( s, i, s[i, j])
        PRINT-OPTIMAL-PARENS( s, s[i, j] + 1, j)
        Print ")"

LCS-LENGTH(X, Y,m, n)
    for i ← 1 to m do c[i,
    0] ← 0 for j ← 0 to n
```

do $c[0, j] \leftarrow 0$ **for** i

$\leftarrow 1$ **to** m

do **for** $j \leftarrow 1$ **to** n

do if $x_i = y_j$

then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$

```
                b[i, j ] ← ' \'
        else    if c[i − 1, j ] ≥ c[i, j − 1]
                then c[i, j ] ← c[i − 1, j ]
                     b[i, j ] ←
                       '|'
        else c[i, j ] ← c[i, j − 1]
                b[i, j ] ← '-'
    return c and b

PRINT-LCS(b, X, i, j )
    if i = 0 or j = 0
    then return
    if b[i, j ] = '
    \'
    then PRINT-LCS(b, X, i − 1, j − 1)
    print xi
    else    if b[i, j ] = '|'
            then PRINT-LCS(b, X, i − 1, j )
    else PRINT-LCS(b, X, i, j − 1)

0-1-KNAPSACK(W, value[ ], weight[ ])
    for w = 0 to W
            B[0,w] =
            0
    for i = 0 to n
            B[i,0] = 0
    for i = 0 to n
            for w = 0 to W
                    if weight_i <= w
                            if value_i + B[i-1,w-w_i] > B[i-1,w]
                                    B[i,w] = value_i + B[i-1,w-
                                    w_i]
```

```
                    else
                                    B[i,w] = B[i-1,w]
                                     else B[i,w] = B[i-1,w]
return B
```

Department of Computer Science & Engineering

Input:

Matrix Chain Multiplication Problem: Elements of two 3X3 matrix

LCS Problem: Two strings

0-1 Knapsack Problem: Capacity of knapsack, number of elements and their weights.

Output:

Matrix Chain Multiplication Problem: Product of two 3X3 matrix

LCS Problem: LCS of the input strings

Knapsack Problem: Optimal solution containing the items selected.

Post-Experiment Questions:

Q1.How would you write a Brute Force Program for the above algorithms? Q2.What is the running time of each of the algorithm implemented? Q3.Compare the running time with Brute Force algorithms.

Department of Computer Science & Engineering

References

1. Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, “Introduction to Algorithms”, Printice Hall of India.

2. E. Horowitz & S Sahni, "Fundamentals of Computer Algorithms", Second edition.

3. Aho, Hopcraft, Ullman, “The Design and Analysis of Computer Algorithms” Pearson Education, 2008.

Department of Computer Science & Engineering

APPENDIX

AKTU SYLLABUS

RCS-552 Design and Analysis of Algorithm Lab

Objective:-

1. Program for Recursive Binary &Linear Search.
2. Program for Heap Sort.
3. Program for Merge Sort.
4. Program for Selection Sort.
5. Program for Insertion Sort.
6. Program for Quick Sort.
7. Knapsack Problem using Greedy Solution
8. Perform Travelling Salesman Problem
9. Find Minimum Spanning Tree using Kruskal's Algorithm
10. Implement N Queen Problem using Backtracking