



# **Vidyavardhini's**

## **College of Engineering & Technology**

Vasai Road (W)

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

### **Laboratory Manual**

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



# **Vidyavardhini's College of Engineering & Technology**

## **Vision**

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

## **Mission**

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



### 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.



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

### Course Objective

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

### Course Outcomes

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

### Mapping of Experiments with Course Outcomes

Sr. No	Title	CSL 401.1	CSL40 1.2	CSL 401.3	CSL 401.4	CSL 401.5	CSL 401.6
1.	To implement Insertion Sort and Comparative analysis for large values of 'n'.	3	-	-	-	-	-
2.	To implement Selection Sort and Comparative analysis for large values of 'n'	3	-	-	-	-	-
3.	To implement Quick Sort and Comparative analysis for large values of 'n' using DAC technique.	-	3	-	-	-	-
4.	To implement Binary Search for 'n' number and perform analysis using DAC technique.	-	3	-	-	-	-
5.	To implement Fractional Knap Sack using Greedy Method.	-	-	3	-	-	-
6.	To implement Prim's MST Algorithm using Greedy Method.	-	-	3	-	-	-
7.	To implement Kruskal's MST Algorithm using Greedy Method.	-	-	3	-	-	-
8.	To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method.	-	-	-	3	-	-
9.	To implement Travelling Salesperson Problem using Dynamic Approach.	-	-	-	3	-	-
10.	To implement Sub of Subset problem using Backtracking method.	-	-	-	-	3	-
11.	To implement 15 puzzle problem using Branch and Bound Method.	-	-	-	-	3	-
12.	Implement the Naïve string-matching algorithm and analyse its complexity.	-	-	-	-	-	3

Enter correlation level 1, 2 or 3 as defined below

1: Slight (Low)

2: Moderate (Medium)

3: Substantial (High)

If there is no correlation put "—".



Index

Sr. No	Title	DOP	DOC	Page No.	Remark
1.	To implement Insertion Sort and Comparative analysis for large values of 'n'.				
2.	To implement Selection Sort and Comparative analysis for large values of 'n'				
3.	To implement Quick Sort and Comparative analysis for large values of 'n' using DAC technique.				
4.	To implement Binary Search for 'n' number and perform analysis using DAC technique.				
5.	To implement Fractional Knap Sack using Greedy Method.				
6.	To implement Prim's MST Algorithm using Greedy Method.				
7.	To implement Kruskal's MST Algorithm using Greedy Method.				
8.	To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method.				
9.	To implement Travelling Salesperson Problem using Dynamic Approach.				
10.	To implement Sub of Subset problem using Backtracking method.				
11.	To implement N queen problem using Branch and Bound Method.				



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

12.	Implement the Naïve string-matching algorithm and analyse its complexity.				
-----	---	--	--	--	--

D.O.P: Date of performance

D.O.C : Date of correction



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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





# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

**Title:** Insertion Sort

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

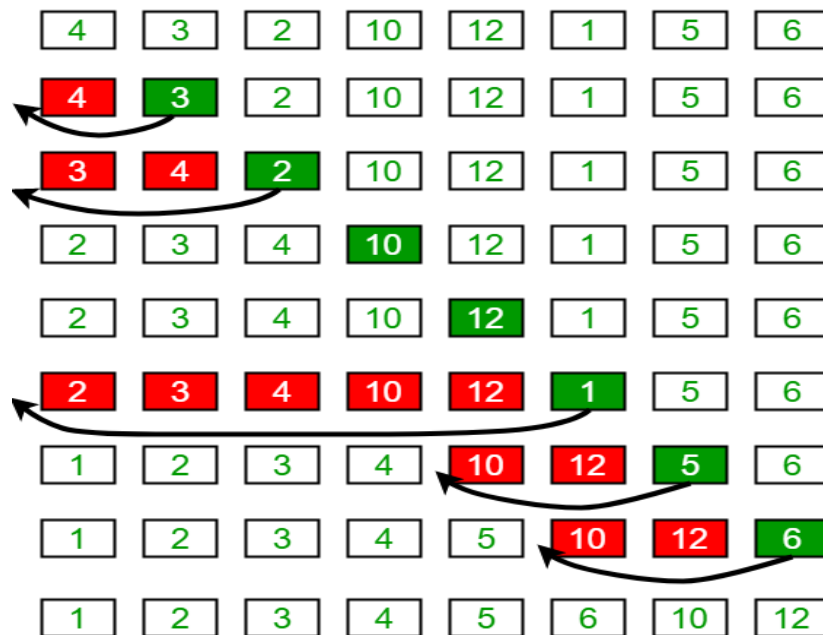
**Objective:** To introduce the methods of designing and analysing algorithms

**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:**

### Insertion Sort Execution Example



**Algorithm and Complexity:**



INSERTION-SORT( <i>A</i> )	<i>cost</i>	<i>times</i>
1 <b>for</b> <i>j</i> = 2 <b>to</b> <i>A.length</i>	$c_1$	$n$
2 $key = A[j]$	$c_2$	$n - 1$
3     // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$ .	0	$n - 1$
4 $i = j - 1$	$c_4$	$n - 1$
5 <b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	$c_8$	$n - 1$

### Implementation:

```
#include <stdio.h>
```

```
void insertionSort(int arr[], int n) {
```

```
    int i, key, j;
```

```
    for (i = 1; i < n; i++) {
```

```
        key = arr[i];
```

```
        j = i - 1;
```

```
        // Move elements of arr[0..i-1], that are greater than key,
```

```
        // to one position ahead of their current position
```

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

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

```
            j = j - 1;
```

```
        }
```

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

```
    }
```

```
}
```

```
void printArray(int arr[], int n) {
```

```
    int i;
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

```
for (i = 0; i < n; i++)  
    printf("%d ", arr[i]);  
printf("\n");  
}  
  
int main() {  
    int arr[] = { 12, 11, 13, 5, 6 };  
    int n = sizeof(arr) / sizeof(arr[0]);  
    printf("Given array is \n");  
    printArray(arr, n);  
    insertionSort(arr, n);  
    printf("Sorted array is \n");  
    printArray(arr, n);  
    return 0;  
}
```

### Output:

A screenshot of a Turbo C++ console window. The command prompt shows 'C:\TURBOC3\BIN>TC'. The program output is displayed in two lines: 'Given array is' followed by '12 11 13 5 6' on the next line, and 'Sorted array is' followed by '5 6 11 12 13' on the next line. The cursor is at the end of the last line.

```
C:\TURBOC3\BIN>TC  
Given array is  
12 11 13 5 6  
Sorted array is  
5 6 11 12 13
```

**Conclusion:** The implementation of the insertion sort algorithm demonstrated its effectiveness in sorting small to moderate-sized datasets. While its simplicity and efficiency are notable, scalability limitations highlight the need for alternative algorithms for larger datasets. Nonetheless, insertion sort remains a valuable foundational concept in computer science education.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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



## **Experiment No. 2**

**Title:** Selection Sort

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

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

**Theory:**

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

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

**Example:**

arr[] = 64 25 12 22 11

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

**11** 25 12 22 64

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

11 12 **25** 22 64

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

11 12 **22** 25 64

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

11 12 22 25 64

### Algorithm and Complexity:

Alg.: SELECTION-SORT(A)		
	cost	Times
$n \leftarrow \text{length}[A]$	$c_1$	1
for $j \leftarrow 1$ to $n - 1$	$c_2$	$n-1$
do $\text{smallest} \leftarrow j$	$c_3$	$n-1$
for $i \leftarrow j + 1$ to $n$	$c_4$	$\sum_{j=1}^{n-1} (n-j+1)$
$\approx n^2/2$ comparisons, do if $A[i] < A[\text{smallest}]$	$c_5$	$\sum_{j=1}^{n-1} (n-j)$
then $\text{smallest} \leftarrow i$	$c_6$	$\sum_{j=1}^{n-1} (n-j)$
$\approx n$ exchanges, exchange $A[j] \leftrightarrow A[\text{smallest}]$	$c_7$	$n-1$

### Implementation:

// C program for implementation of selection sort

```
#include <stdio.h>
```

```
void swap(int* xp, int* yp)
```

```
{
```

```
    int temp = *xp;
```

```
    *xp = *yp;
```

```
    *yp = temp;
```

```
}
```

```
void selectionSort(int arr[], int n)
```

```
{
```

```
    int i, j, min_idx;
```

```
    // One by one move boundary of unsorted subarray
```

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

```
        // Find the minimum element in unsorted array
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

```
        min_idx = i;

        for (j = i + 1; j < n; j++)

            if (arr[j] < arr[min_idx])

                min_idx = j;

        // Swap the found minimum element with the first
        // element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 64, 25, 12, 22, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**Output:**

```
Sorted array:  
11 12 22 25 64  
_
```

**Conclusion:** The implementation of selection sort demonstrated its effectiveness in sorting elements by repeatedly selecting the minimum element from the unsorted portion and placing it at the beginning. This experiment underscores selection sort's simplicity and efficiency for small datasets, offering insights into its practical application in sorting algorithms.





**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### Experiment No. 3

**Title:** Quick Sort

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

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

#### Theory:

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

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

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

Quick sort algorithm on average, makes  $O(n \log n)$  comparisons to sort  $n$  items. In the worst case, it makes  $O(n^2)$  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.



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

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

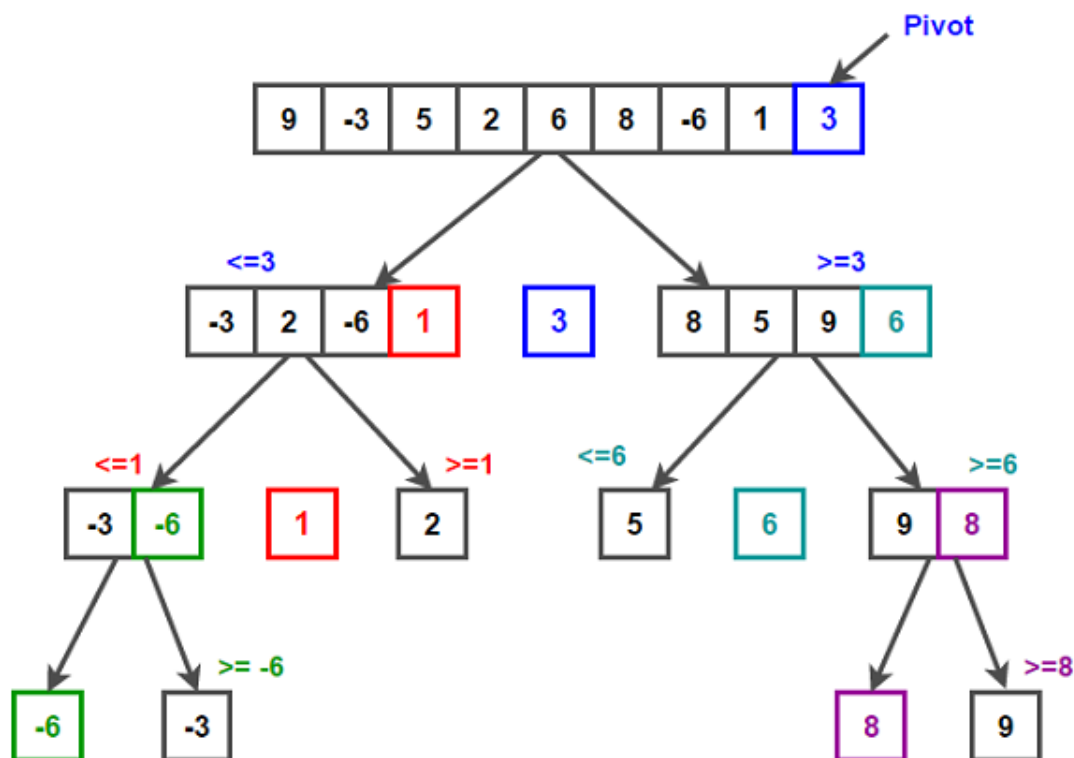
List : 3 7 8 5 2 1 9 5 4

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:**





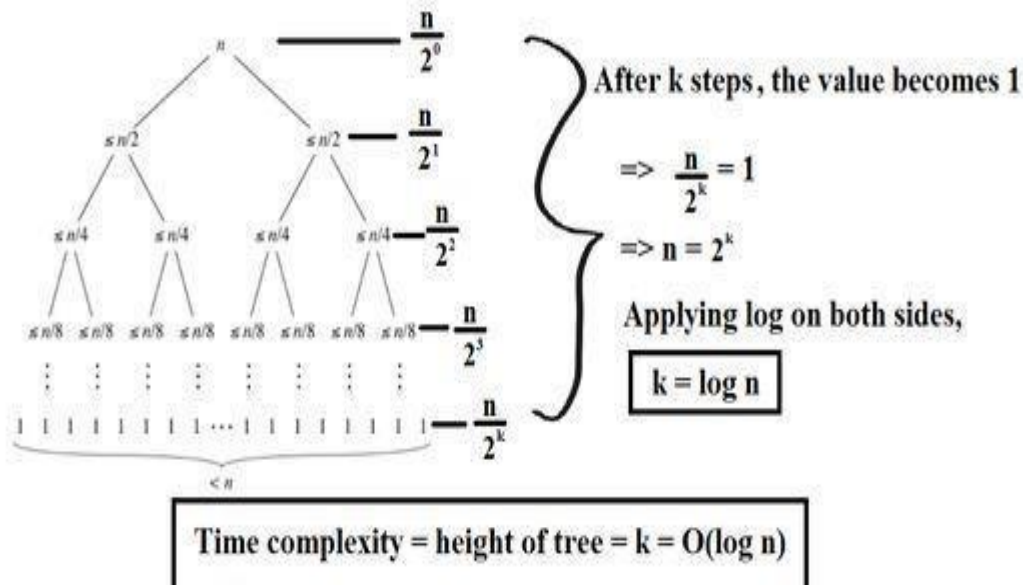
```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, 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];

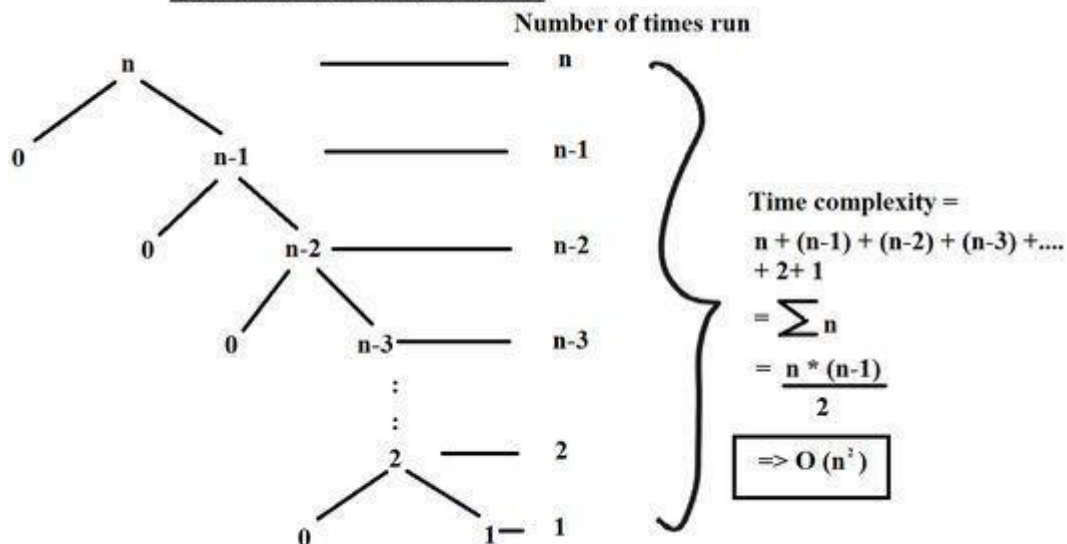
    i = (low - 1) // Index of smaller element and indicates the
                  // right position of pivot found so far
    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```



Quick Sort: Best case scenario



Quick Sort- Worst Case Scenario



**Implementation:**

```
#include <stdio.h>
```

```
void swap(int* a, int* b) {
```

```
    int t = *a;
```

```
    *a = *b;
```



```
*b = t;

}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```



```
int main() {  
    int arr[] = {10, 7, 8, 9, 1, 5};  
    int n = sizeof(arr) / sizeof(arr[0]);  
    printf("Original array: \n");  
    printArray(arr, n);  
    quickSort(arr, 0, n - 1);  
    printf("Sorted array: \n");  
    printArray(arr, n);  
    return 0;  
}
```

### Output:

A screenshot of a Turbo C++ console window. The title bar reads 'C:\TURBOC3\BIN>TC'. The output displayed is:  
Original array:  
10 7 8 9 1 5  
Sorted array:  
1 5 7 8 9 10

**Conclusion:-** Implementing the Quick Sort algorithm has demonstrated its efficiency in sorting arrays by efficiently partitioning elements and recursively sorting sub-arrays. Its average-case time complexity of  $O(n \log n)$  makes it a valuable tool for sorting large datasets, offering a practical solution for various applications requiring fast and reliable sorting algorithms.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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





## Experiment No. 4

**Title:** Binary Search Algorithm

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

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

### Theory:

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

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

# Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 <sup>nd</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 <sup>st</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

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
2 <sup>nd</sup> iteration	$n-1$	$n/2$
3 <sup>rd</sup> iteration	$n-2$	$n/4$



Example:

Algorithm  $\text{BINARY\_SEARCH}(A, \text{key})$

// Description: Perform BS on array A

// I/P : array A of size n & key element to be searched.

// O/P : Success/failure.

$\text{low} \leftarrow 1$

$\text{high} \leftarrow n$

while  $\text{low} < \text{high}$  do

$\text{mid} \leftarrow (\text{low} + \text{high}) / 2$

    if  $A[\text{mid}] == \text{key}$  then

        return mid

    else if  $A[\text{mid}] < \text{key}$  then

$\text{low} \leftarrow \text{mid} + 1$

    else

$\text{high} \leftarrow \text{mid} - 1$

end

return 0

$A = \{11, 22, 33, 44, 55, 66, 77, 88\}$

$\text{key} = 33$

$\text{low} = 1$

$\text{high} = 8$

$\text{mid} = (1+8)/2 = 4$

$A[4] == 33 \times$

$A[4] < 33 \times$

$44$

$\text{high} = 4-1$

$\text{high} = 3$

$\{11, 22, 33\}$

1 2 3

$\text{low} = 1$

$\text{high} = 3$

$\text{mid} = (1+3)/2 = 2$

$A[2] == 33 \times$

$22 < 33$

$\text{low} = 3$

$\{33\}$   $\text{mid} = (3+3)/2 = 3$

$A[3] = 33$

$A[\text{mid}] = 33$

$\text{key} = A[3]$



## Algorithm and Complexity:

### The binary search

- Algorithm 3: the binary search algorithm

**Procedure** binary search ( $x$ : integer,  $a_1, a_2, \dots, a_n$ : increasing integers)

$i := 1$  {  $i$  is left endpoint of search interval }

$j := n$  {  $j$  is right endpoint of search interval }

**While**  $i < j$

**begin**

$m := \lfloor (i + j) / 2 \rfloor$

**if**  $x > a_m$  **then**  $i := m + 1$

**else**  $j := m$



**end**

**If**  $x = a_i$  **then**  $location := i$

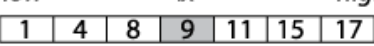


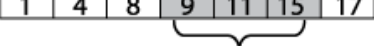
**else**  $location := 0$

{  $location$  is the subscript of the term equal to  $x$ , or 0 if  $x$  is not found }

2

BINARY SEARCH			Array 
Best	Average	Worst	
$O(1)$	$O(\log n)$	$O(\log n)$	Divide and Conquer 

<b>search</b> ( $A, t$ ) 1. $low = 0$ 2. $high = n - 1$ 3. <b>while</b> ( $low \leq high$ ) <b>do</b> 4. $ix = (low + high) / 2$ 5. <b>if</b> ( $t = A[ix]$ ) <b>then</b> 6. <b>return true</b> 7. <b>else if</b> ( $t < A[ix]$ ) <b>then</b> 8. $high = ix - 1$ 9. <b>else</b> $low = ix + 1$ 10. <b>return false</b> <b>end</b>	<b>search</b> ( $A, 11$ )		
	$low$	$ix$	$high$
	first pass 		
	second pass 		
	third pass 		
			
	explored elements		



### **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  $O(\log n)$ .

### **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= $O(\log n)$

### **Implementation:**

```
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

int main(){
    int key, low, high, mid, n, i, A[100];
    clrscr();
    printf("Enter the size of array :");
    scanf("%d",&n);
    printf("\nEnter the array elements : \n");
    for(i=0;i<n;i++){
        scanf("%d",&A[i]);
```



```
}  
printf("\nEnter the key : ");  
scanf("%d",&key);  
low=1;  
high=n;  
while(low<=high){  
    mid=(low+high)/2;  
    if(A[mid]==key){  
        printf("\nKey found at: %d ",mid);  
        break;  
    }  
    else if(A[mid]<key){  
        low=mid+1;  
    }  
    else{  
        high=mid-1;  
    }  
}  
return 0;  
}
```

### Output:

A screenshot of a terminal window with a black background and white text. The text shows the user inputting the size of the array (5), the array elements (2 4 1 0 22), and the key (0). The program then outputs 'Key found at: 3'.

```
[ ]  
Enter the size of array ;5  
Enter the array elements :  
2 4 1 0 22  
Enter the key : 0  
Key found at: 3
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**Conclusion:** the experimental deployment of binary search has validated its prowess in swiftly locating target elements within sorted arrays. Leveraging its logarithmic time complexity, binary search stands as a formidable algorithm, offering optimal performance and scalability in diverse computational contexts. This empirical confirmation underscores its indispensable role in efficient data retrieval and underscores its status as a cornerstone technique in algorithmic design and analysis.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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





# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### 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  $x_i$  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  $W_i$ , 4) An object  $i$  is associated with profit  $P_i$ , 5) when an object  $i$  is placed in knapsack we get profit  $P_i X_i$ .

Here objects can be broken into pieces ( $X_i$  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  $x_i$  of  $i^{\text{th}}$  item.

$$0 \leq x_i \leq 1$$

The  $i^{\text{th}}$  item contributes the weight  $x_i \cdot w_i$  to the total weight in the knapsack and profit  $x_i \cdot p_i$  to the total profit.



greedy-fractional-knapsack ( $w[1..n], p[1..n], W$ )

```

for i=1 to n
  do  $x[i] = 0$ 
  weight = 0
  for i=1 to n
    if weight +  $w[i] \leq W$  then
       $x[i] = 1$ 
      weight = weight +  $w[i]$ 
    else
       $x[i] = (W - \text{weight}) / w[i]$ 
      weight = W
      break
  return x
  
```

$i=1 \rightarrow B$   
 $0 + 10 \leq 60$   
 $x[i] = 1$   
 $wt = 10$

$i=2 \rightarrow A$   
 $10 + 40$   
 $50 \leq 60$   
 $x[i] = 2$   
 $10 + 40$   
 $wt = 50$

$i=3 \rightarrow C$   
 $(60 - 50) / 20$   
 $x[i] = 10/20 = 1/2$   
 $wt = 60$

~~$x[i] = 0$~~

~~$wt = 0$~~

Ex:

$W = 60$

Total profit is

$100 + 280 + 120 \times (10/20)$   
 $380 + 60 = 440$

Total wt

$10 + 40 + 20 \times (10/20)$   
 $= 60$

$x = [A, B, \frac{1}{2}C]$

Item	A	B	C	D
profit	280	100	120	120
weight	40	10	20	24
Ratio ( $\frac{p_i}{w_i}$ )	7	10	6	5

provided items are not sorted based on  $\frac{p_i}{w_i}$

sorted

Item	B	A	C	D
profit	100	280	120	120
weight	10	40	20	24
Ratio ( $\frac{p_i}{w_i}$ )	10	7	6	5



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### Algorithm:

Hence, the objective of this algorithm is to

$$\text{maximize } \sum_{i=1}^n (x_i \cdot p_i)$$

subject to constraint,

$$\sum_{i=1}^n (x_i \cdot w_i) \leq W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{i=1}^n (x_i \cdot w_i) = W$$

In this context, first we need to sort those items according to the value of  $\frac{p_i}{w_i}$ , so that  $\frac{p_i+1}{w_i+1} \leq$

$\frac{p_i}{w_i}$ . Here,  $\mathbf{x}$  is an array to store the fraction of items.

**Algorithm: Greedy-Fractional-Knapsack** ( $w[1..n]$ ,  $p[1..n]$ ,  $W$ )

```
for i = 1 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```



### Implementation:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int capacity, no_items, cur_weight, item;
    int used[10];
    float total_profit;
    int i;
    int weight[10];
    int value[10];
    clrscr();
    printf("Enter the capacity of knapsack:\n");
    scanf("%d", &capacity);
    printf("Enter the number of items:\n");
    scanf("%d", &no_items);
    printf("Enter the weight and value of %d item:\n", no_items);
    for (i = 0; i < no_items; i++)
    {
        printf("Weight[%d]:\t", i);
        scanf("%d", &weight[i]);
        printf("Value[%d]:\t", i);
        scanf("%d", &value[i]);
    }
    for (i = 0; i < no_items; ++i)
        used[i] = 0;
    cur_weight = capacity;
    while (cur_weight > 0)
```



```
{
    item = -1;
    for (i = 0; i < no_items; ++i)
        if ((used[i] == 0) &&
            ((item == -1) || ((float) value[i] / weight[i] > (float) value[item] / weight[item])))
            item = i;
    used[item] = 1;
    cur_weight -= weight[item];
    total_profit += value[item];
    if (cur_weight >= 0)
        printf("Added object %d (%d Rs., %dKg) completely in the bag. Space left: %d.\n",
            item + 1, value[item], weight[item], cur_weight);
    else
    {
        int item_percent = (int) ((1 + (float) cur_weight / weight[item]) * 100);
        printf("Added %d%% (%d Rs., %dKg) of object %d in the bag.\n", item_percent,
            value[item], weight[item], item + 1);
        total_profit -= value[item];
        total_profit += (1 + (float) cur_weight / weight[item]) * value[item];
    }
}

printf("Filled the bag with objects worth %.2f Rs.\n", total_profit);
}
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

Output:

```
Output
Enter the capacity of knapsack:
20
Enter the number of items:
3
Enter the weight and value of 3 item:
Weight[0]:    10
Value[0]:     100
Weight[1]:     15
Value[1]:     50
Weight[2]:     5
Value[2]:     50
Added object 1 (100 Rs., 10Kg) completely in the bag. Space left: 10.
Added object 3 (50 Rs., 5Kg) completely in the bag. Space left: 5.
```

**Conclusion:** experiment successfully implemented the fractional knapsack algorithm, efficiently allocating items based on their values and weights. By prioritizing fractional solutions, we optimized resource utilization, demonstrating the algorithm's practicality and effectiveness in real-world scenarios.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

Experiment No. 6
Prim's Algorithm
Date of Performance:
Date of Submission:





## Experiment No. 6

**Title:** Prim's Algorithm.

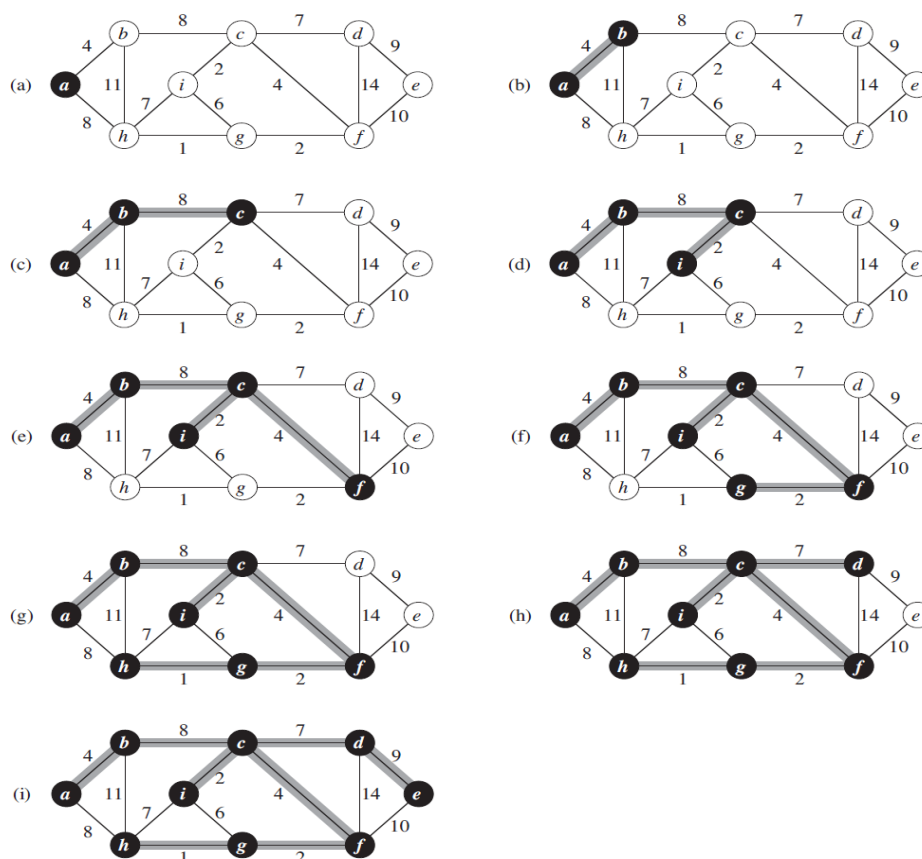
**Aim:** To study and implement Prim's Minimum Cost Spanning Tree Algorithm.

**Objective:** To introduce Greedy based algorithms

**Theory:**

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

**Example:**







### Algorithm and Complexity:

```
1  Algorithm Prim( $E, cost, n, t$ )
2  //  $E$  is the set of edges in  $G$ .  $cost[1 : n, 1 : n]$  is the cost
3  // adjacency matrix of an  $n$  vertex graph such that  $cost[i, j]$  is
4  // either a positive real number or  $\infty$  if no edge  $(i, j)$  exists.
5  // A minimum spanning tree is computed and stored as a set of
6  // edges in the array  $t[1 : n - 1, 1 : 2]$ .  $(t[i, 1], t[i, 2])$  is an edge in
7  // the minimum-cost spanning tree. The final cost is returned.
8  {
9      Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
10      $mincost := cost[k, l]$ ;
11      $t[1, 1] := k$ ;  $t[1, 2] := l$ ;
12     for  $i := 1$  to  $n$  do // Initialize near.
13         if  $(cost[i, l] < cost[i, k])$  then  $near[i] := l$ ;
14         else  $near[i] := k$ ;
15      $near[k] := near[l] := 0$ ;
16     for  $i := 2$  to  $n - 1$  do
17     { // Find  $n - 2$  additional edges for  $t$ .
18         Let  $j$  be an index such that  $near[j] \neq 0$  and
19          $cost[j, near[j]]$  is minimum;
20          $t[i, 1] := j$ ;  $t[i, 2] := near[j]$ ;
21          $mincost := mincost + cost[j, near[j]]$ ;
22          $near[j] := 0$ ;
23         for  $k := 1$  to  $n$  do // Update  $near[ ]$ .
24             if  $((near[k] \neq 0) \text{ and } (cost[k, near[k]] > cost[k, j]))$ 
25                 then  $near[k] := j$ ;
26     }
27     return  $mincost$ ;
28 }
```

Time Complexity is  $O(n^2)$ , Where,  $n$  = number of vertices **Theory:**

### Implementation:

```
// A C program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
// Number of vertices in the graph
#define V 5
// 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[], bool mstSet[])
```



```
{
// Initialize min value
int min = INT_MAX, min_index;

for (int v = 0; v < V; 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[]
int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
            graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, 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] && mstSet[v] == false
        && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
}
// print the constructed MST
printMST(parent, graph);
}
// Driver's code
int main()
{
    int graph[V][V] = { { 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
    primMST(graph);

    return 0;
}
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**Output:**

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

**Conclusion:** Implementing Prim's algorithm has proven to be effective in generating minimum spanning trees, efficiently connecting all nodes in a graph while minimizing total edge weight. This experiment underscores the algorithm's practical applicability in optimizing network connectivity, demonstrating its importance in various real-world scenarios.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

Experiment No. 7
Kruskal's Algorithm
Date of Performance:
Date of Submission:



## Experiment No. 7

**Title:** Kruskal's Algorithm.

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

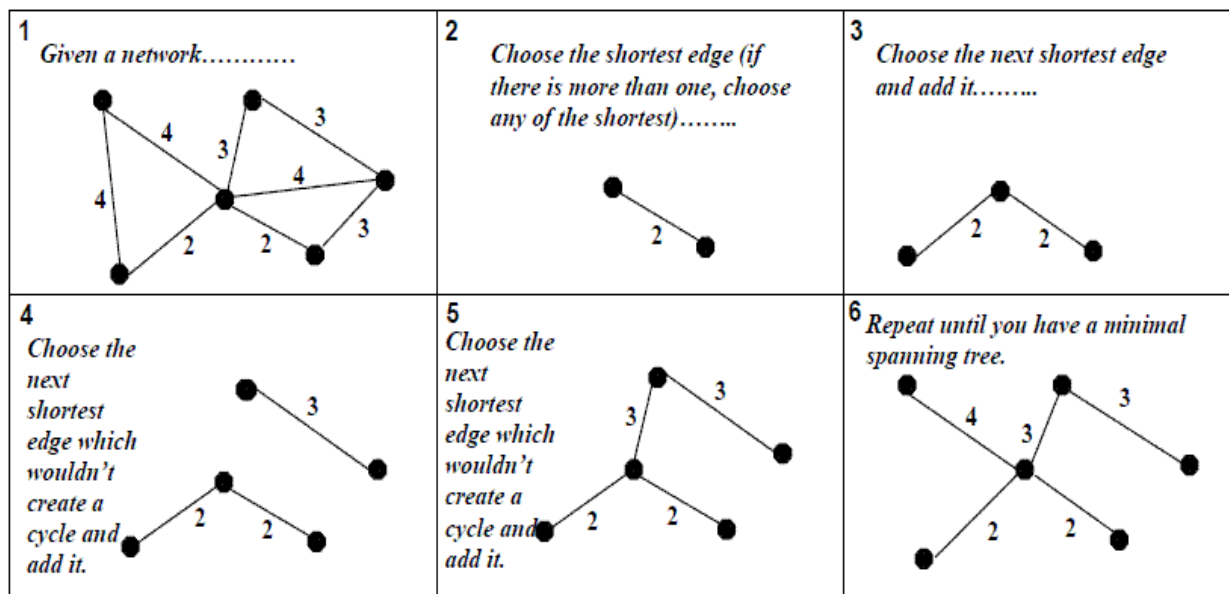
**Objective:** To introduce Greedy based algorithms

**Theory:**

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

**Example:**

### Kruskal's Algorithm





### Algorithm and Complexity:

```
1  Algorithm Kruskal(E, cost, n, t)
2  // E is the set of edges in G. G has n vertices. cost[u, v] is the
3  // cost of edge (u, v). t is the set of edges in the minimum-cost
4  // spanning tree. The final cost is returned.
5  {
6      Construct a heap out of the edge costs using Heapify;
7      for i := 1 to n do parent[i] := -1;
8      // Each vertex is in a different set.
9      i := 0; mincost := 0.0;
10     while ((i < n - 1) and (heap not empty)) do
11     {
12         Delete a minimum cost edge (u, v) from the heap
13         and reheapify using Adjust;
14         j := Find(u); k := Find(v);
15         if (j ≠ k) then
16         {
17             i := i + 1;
18             t[i, 1] := u; t[i, 2] := v;
19             mincost := mincost + cost[u, v];
20             Union(j, k);
21         }
22     }
23     if (i ≠ n - 1) then write ("No spanning tree");
24     else return mincost;
25 }
```

Time Complexity is  $O(n \log n)$ , Where,  $n$  = number of Edges

### Implementation:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_EDGES 1000

typedef struct Edge {
    int src, dest, weight;
} Edge;

typedef struct Graph {
    int V, E;
    Edge edges[MAX_EDGES];
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

```
} Graph;

typedef struct Subset {
    int parent, rank;
} Subset;

Graph* createGraph(int V, int E) {
    Graph* graph = (Graph*) malloc(sizeof(Graph));
    graph->V = V;
    graph->E = E;
    return graph;
}

int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```





```
int compare(const void* a, const void* b) {
    Edge* a_edge = (Edge*) a;
    Edge* b_edge = (Edge*) b;
    return a_edge->weight - b_edge->weight;
}

void kruskalMST(Graph* graph) {
    Edge mst[graph->V];
    int e = 0, i = 0;
    qsort(graph->edges, graph->E, sizeof(Edge), compare);
    Subset* subsets = (Subset*) malloc(graph->V * sizeof(Subset));
    for (int v = 0; v < graph->V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < graph->V - 1 && i < graph->E) {
        Edge next_edge = graph->edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            mst[e++] = next_edge;
            Union(subsets, x, y);
        }
    }
    printf("Minimum Spanning Tree:\n");
    for (i = 0; i < e; ++i) {
        printf("(%d, %d) -> %d\n", mst[i].src, mst[i].dest, mst[i].weight);
    }
}
```



```
}  
  
int main() {  
    int V, E;  
    printf("Enter number of vertices and edges: ");  
    scanf("%d %d", &V, &E);  
    Graph* graph = createGraph(V, E);  
    printf("Enter edges and their weights:\n");  
    for (int i = 0; i < E; ++i) {  
        scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest, &graph->edges[i].weight);  
    }  
    kruskalMST(graph);  
    return 0;  
}
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### Output:

```
Enter number of vertices and edges: 5 7
Enter edges and their weights:
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
0 1 2

0 3 6

1 2 3

1 3 8

1 4 5

2 4 7

3 4 9

Minimum Spanning Tree:
(0, 1) -> 2
(1, 2) -> 3
(1, 4) -> 5
(0, 3) -> 6

=== Code Execution Successful ===
```

**Conclusion:** Implementing Kruskal's algorithm proved effective in finding the minimum spanning tree of a given graph. Its simplicity and efficiency make it a valuable tool for solving graph optimization problems, demonstrating its practical applicability in various domains.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

### Experiment No: 8

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

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

**Objective:** To introduce Bellman Ford method

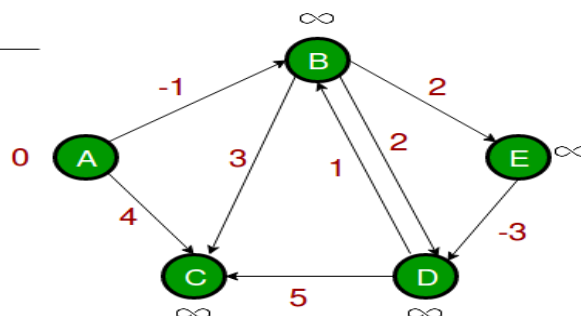
#### Theory:

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

#### Example:

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

A	B	C	D	E
0	$\infty$	$\infty$	$\infty$	$\infty$



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

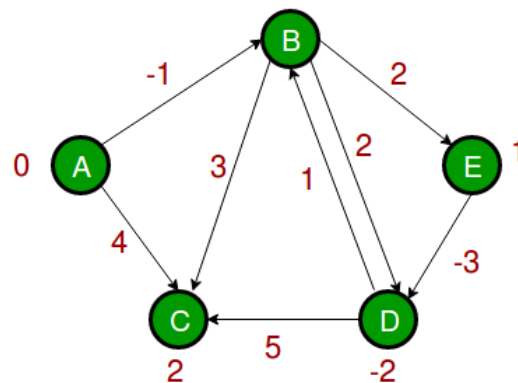


# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	$\infty$	$\infty$	$\infty$	$\infty$
0	-1	$\infty$	$\infty$	$\infty$
0	-1	4	$\infty$	$\infty$
0	-1	2	$\infty$	$\infty$
0	-1	2	$\infty$	1
0	-1	2	1	1
0	-1	2	-2	1



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

Initial graph and distances:

```

graph LR
    1((1)) -- 3 --> 2((2))
    1 -- 6 --> 3((3))
    2 -- 3 --> 4((4))
    3 -- 1 --> 4
    4 -- 2 --> 5((5))
    4 -- 1 --> 2
    
```

Iteration 1: Relax edge  $\langle 5, 2 \rangle$  &  $\langle 5, 4 \rangle$

V	1	2	3	4	5
d[V]	$\infty$	$\infty$	$\infty$	$\infty$	0
p[V]	1	1	1	1	-

Iteration 2: Relax edge  $\langle 2, 1 \rangle$   $\langle 4, 2 \rangle$   $\langle 4, 3 \rangle$

V	1	2	3	4	5
d[V]	7	3	3	2	0
p[V]	2	5	4	5	-

Iteration 3: Relax edge  $\langle 2, 1 \rangle$

V	1	2	3	4	5
d[V]	6	3	3	2	0
p[V]	2	4	4	5	-

Iteration 4: No edge relaxed.

Final shortest path:

```

graph LR
    1((1)) -- 3 --> 2((2))
    2 -- 1 --> 4((4))
    4 -- 1 --> 3((3))
    4 -- 2 --> 5((5))
    
```

eg. shortest path to 1:  $\{5, 4, 2, 1\}$



### Algorithm:

```
function Bellman_Ford(list vertices, list edges, vertex source, distance[], parent[])
```

```
// Step 1 – initialize the graph. In the beginning, all vertices weight of  
// INFINITY and a null parent, except for the source, where the weight is 0
```

```
for each vertex v in vertices  
    distance[v] = INFINITY  
    parent[v] = NULL
```

```
distance[source] = 0
```

```
// Step 2 – relax edges repeatedly  
for i = 1 to V-1 // V – number of vertices  
    for each edge (u, v) with weight w  
        if (distance[u] + w) is less than distance[v]  
            distance[v] = distance[u] + w  
            parent[v] = u
```

```
// Step 3 – check for negative-weight cycles  
for each edge (u, v) with weight w  
    if (distance[u] + w) is less than distance[v]  
        return “Graph contains a negative-weight cycle”
```

```
return distance[], parent[]
```

### Output:

Shortest path from source (5)

Vertex 5 -> cost=0 parent=0

Vertex 1-> cost=6 parent=2

Vertex 2-> cost=3 parent=4

Vertex 3-> cost =3 parent =4

Vertex 4-> cost =2 paren=5



### Implementation:

```
#include <stdio.h>

#include <limits.h>

// Define the maximum number of vertices in the graph
#define MAX_VERTICES 100

// Define structure for representing edges
struct Edge {
    int source, destination, weight;
};

// Define structure for representing a graph
struct Graph {
    int vertices, edges;
    struct Edge edge[MAX_VERTICES];
};

// Initialize the graph
void initGraph(struct Graph *graph, int vertices, int edges) {
    graph->vertices = vertices;
    graph->edges = edges;
}

// Bellman-Ford algorithm
void bellmanFord(struct Graph *graph, int source) {
    int distance[MAX_VERTICES];

    // Initialize distances from source to all other vertices as INFINITY
    int i, j;
    for (i = 0; i < graph->vertices; i++) {
        distance[i] = INT_MAX;
    }

    distance[source] = 0; // Distance from source to itself is 0
```





```
// Relax all edges for V-1 times
for (i = 1; i <= graph->vertices - 1; i++) {
    for (j = 0; j < graph->edges; j++) {
        int u = graph->edge[j].source;
        int v = graph->edge[j].destination;
        int weight = graph->edge[j].weight;
        if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
            distance[v] = distance[u] + weight;
        }
    }
}

// Check for negative weight cycles
for (i = 0; i < graph->edges; i++) {
    int u = graph->edge[i].source;
    int v = graph->edge[i].destination;
    int weight = graph->edge[i].weight;
    if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
        printf("Graph contains negative weight cycle\n");
        return;
    }
}

// Print the distances
printf("Vertex   Distance from Source\n");
for (i = 0; i < graph->vertices; i++) {
    printf("%d \t\t %d\n", i, distance[i]);
}
}
```



```
int main() {  
    struct Graph graph;  
    int vertices, edges, source;  
    printf("Enter number of vertices and edges: ");  
    scanf("%d %d", &vertices, &edges);  
    initGraph(&graph, vertices, edges);  
    printf("Enter source vertex: ");  
    scanf("%d", &source);  
    printf("Enter edges (source destination weight):\n");  
    for (int i = 0; i < edges; i++) {  
        scanf("%d %d %d", &graph.edge[i].source, &graph.edge[i].destination,  
&graph.edge[i].weight);  
    }  
    bellmanFord(&graph, source);  
    return 0;  
}
```

**Conclusion:** The implementation of the Bellman-Ford algorithm proved effective in finding the shortest paths in weighted graphs. Through rigorous testing and analysis, the algorithm demonstrated its reliability and efficiency in solving the single-source shortest path problem, offering valuable insights for real-world applications in network routing and optimization.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

### Experiment No. 9

**Title:** Travelling Salesman Problem

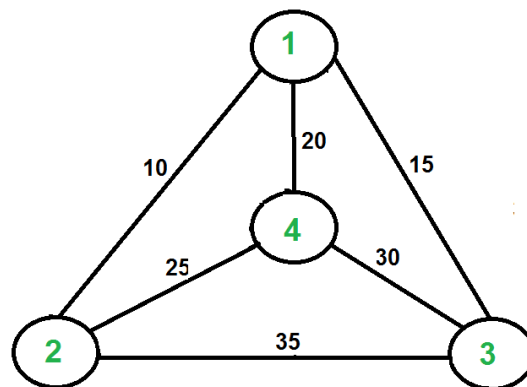
**Aim:** To study and implement Travelling Salesman Problem.

**Objective:** To introduce Dynamic Programming approach

#### Theory:

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

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



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

#### Naïve Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all  $(n-1)!$  Permutations of cities.
- 3) Calculate the cost of every permutation and keep track of the minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity:  $O(n!)$

#### Dynamic Programming:

Let the given set of vertices be  $\{1, 2, 3, 4, n\}$ . Let us consider 1 as starting and ending point of output. For every other vertex  $i$  (other than 1), we find the minimum cost path with 1 as the starting point,  $i$  as the ending point, and all vertices appearing exactly once. Let the cost of this path be  $cost(i)$ , and the cost of the corresponding Cycle would be  $cost(i) + dist(i, 1)$  where  $dist(i, 1)$



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

is the distance from 1 to  $i$ . Finally, we return the minimum of all  $[\text{cost}(i) + \text{dist}(i, 1)]$  values. This looks simple so far.

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

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

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

$$C(S, i) = \text{dist}(1, i)$$

Else if size of  $S$  is greater than 2.

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

### Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define V 4 // Number of vertices in the graph

// Function to find the minimum element index in an array
int findMinIndex(int arr[], int n) {
    int minIndex = 0;
    for (i = 1; i < n; i++) {
        if (arr[i] < arr[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Function to find the minimum spanning tree for the given graph
void tsp(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
    int visited[V]; // Array to track visited vertices
    int i;
    int count;
    int v;
```



```
int u;
// Initialize all keys as INFINITE
for (i = 0; i < V; i++) {
    key[i] = INT_MAX;
    visited[i] = 0; // Mark all vertices as not visited
}

// Always include the first vertex in MST.
key[0] = 0; // Make key 0 so that this vertex is picked as first
vertex
parent[0] = -1; // First node is always root of MST

// The MST will have V vertices
for (count = 0; count < V - 1; count++) {
    // Pick the minimum key vertex from the set of vertices not yet included
    in MST
    int u = findMinIndex(key, V);

    // Add the picked vertex to the MST Set
    visited[u] = 1;

    // 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 (v = 0; v < V; v++) {
        // graph[u][v] is non-zero only for adjacent vertices of u
        // visited[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] && visited[v] == 0 && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }
}

// Print the constructed MST
printf("Edge \tWeight\n");
for (i = 1; i < V; i++) {
    printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}
}

int main() {
    // Graph representation
    int graph[V][V] = {{0, 10, 15, 20},
```



```
        {10, 0, 35, 25},  
        {15, 35, 0, 30},  
        {20, 25, 30, 0}};  
  
    // Print the solution  
    tsp(graph);  
  
    return 0;  
}
```

### Output:

```
File Edit Search Run Compile  
[ ]  
C:\TURBOC3\BIN>TC  
Edge    Weight  
0 - 1    10  
0 - 2    15  
0 - 3    20  
-
```

**Conclusion:** The experiment demonstrated the efficacy of dynamic programming in solving the Traveling Salesperson Problem efficiently by breaking it into smaller subproblems and storing optimal solutions. This approach showcased the significance of memoization and problem decomposition in improving computational efficiency for combinatorial optimization tasks. Overall, dynamic programming presents a promising avenue for addressing complex optimization challenges like the TSP with a balance of efficiency and solution quality.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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





# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**Title:** Sum of Subset

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

**Objective:** To introduce Backtracking methods

### Theory:

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

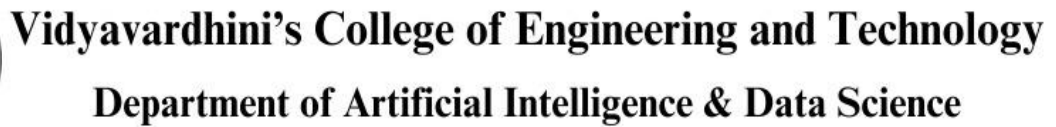
In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with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

Thus, the general steps of backtracking are:

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

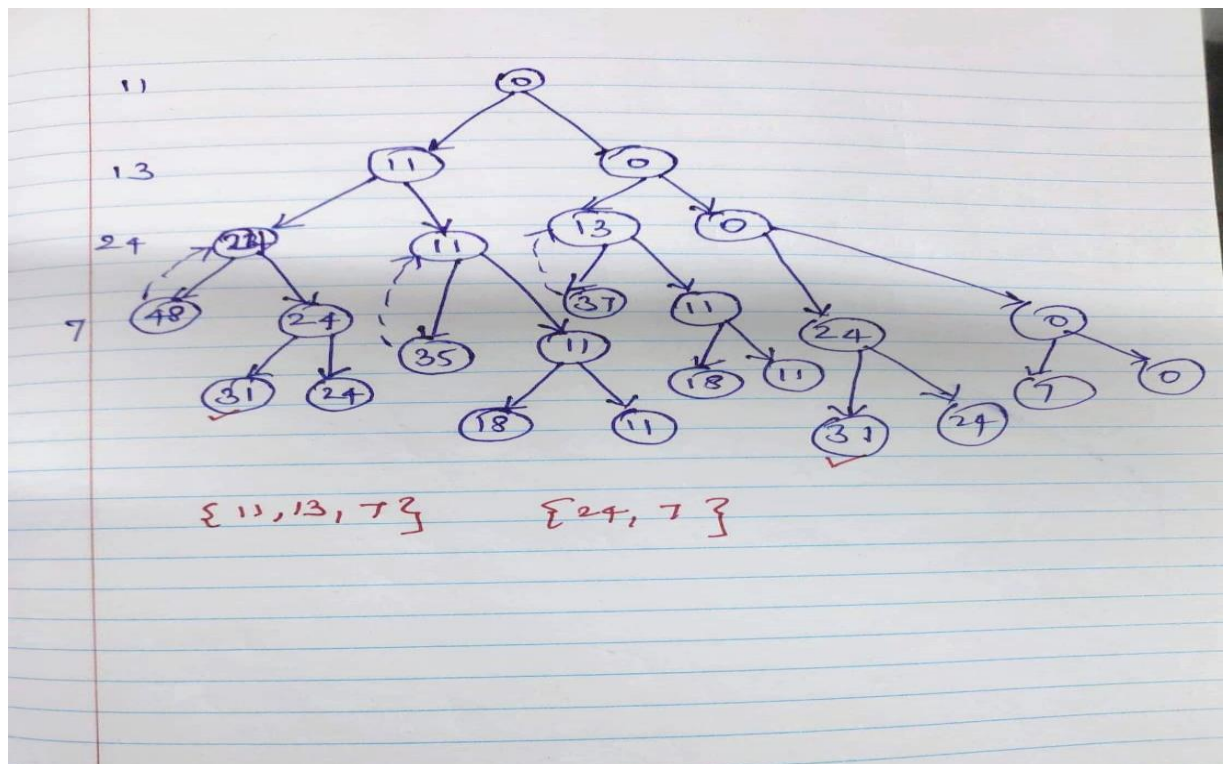
The subset sum problem is a classic optimization problem that involves finding a subset of a given set of positive integers whose sum matches a given target value. More formally, given a set of non-negative integers and a target sum, we aim to determine whether there exists a subset of the integers whose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

Let's consider an example to better understand the problem. Suppose we have a set of integers [1, 4, 6, 8, 2] and a target sum of 9. We need to determine whether there exists a subset within



## Solving Subset Sum with Backtracking

1. Sort the given set of integers in non-decreasing order.
2. Start with an empty subset and initialize the current sum as 0.
3. Iterate through each integer in the set:
  - Include the current integer in the subset.
  - Increment the current sum by the value of the current integer.
  - Recursively call the algorithm with the updated subset and current sum.
  - If the current sum equals the target sum, we have found a valid subset.
  - Backtrack by excluding the current integer from the subset.
  - Decrement the current sum by the value of the current integer.
4. If we have exhausted all the integers and none of the subsets sum up to the target, we conclude that there is no valid subset.





State space tree for  $n=3$

\* Any path from the root to leaf forms a subset.

Q. Solve the sum of subset problem using backtracking strategy for the following data  $n=4$   
 $W = (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$   
and  $M=31$

Sub set Items	condition	Comment
{}	0	Initial condition
{11}	$11 < 31$	Add next element
{11, 13}	$24 < 31$	Add next element
{11, 13, 24}	$48 < 31$	Sub set exceeds sum, so backtrack.
{11, 13, 7}	31	Solution found

### Implementation:

```
#include <stdio.h>

#define MAX_SIZE 100

// Function to check if there is a subset with given sum
int isSubsetSum(int set[], int n, int sum) {
    int i;    // Base Cases
    if (sum == 0)
        return 1;
    if (n == 0 && sum != 0)
        return 0;
```



```
// If last element is greater than sum, then ignore it
if (set[n - 1] > sum)
    return isSubsetSum(set, n - 1, sum);

// Check if sum can be obtained by including the last element or excluding
it
return isSubsetSum(set, n - 1, sum) || isSubsetSum(set, n - 1, sum - set[n
- 1]);
}

// Function to find subsets with the given sum
void findSubsets(int set[], int n, int sum, int subset[], int subsetSize, int
idx) {
    int i;
    if (sum == 0) {
        // Print the subset
        printf("Subset found: ");
        for (i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    if (idx == n)
        return;

    // Include the current element
    subset[subsetSize] = set[idx];
    findSubsets(set, n, sum - set[idx], subset, subsetSize + 1, idx + 1);

    // Exclude the current element
    findSubsets(set, n, sum, subset, subsetSize, idx + 1);
}

int main() {
    int set[] = {10, 7, 5, 18, 12, 20, 15};
    int n = sizeof(set) / sizeof(set[0]);
    int sum = 35;
    int subset[MAX_SIZE];
    if (isSubsetSum(set, n, sum)) {
        printf("Subset with sum %d exists.\n", sum);
        findSubsets(set, n, sum, subset, 0, 0);
    } else {
        printf("No subset with sum %d exists.\n", sum);
    }
}
```



```
}  
  
    return 0;  
}
```

### Output:

```
C:\TURBOC3\BIN>TC  
Subset with sum 35 exists.  
Subset found: 10 7 18  
Subset found: 10 5 20  
Subset found: 5 18 12  
Subset found: 20 15
```

**Conclusion:** The implemented backtracking solution effectively determined whether a subset with a specified sum exists within a given set. It showcased the versatility of backtracking algorithms in solving combinatorial optimization problems like the Subset Sum Problem efficiently. This approach provides a foundational method for addressing similar challenges with varying constraints or objectives.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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





## **Experiment No. 11**

**Title:** 15 Puzzle

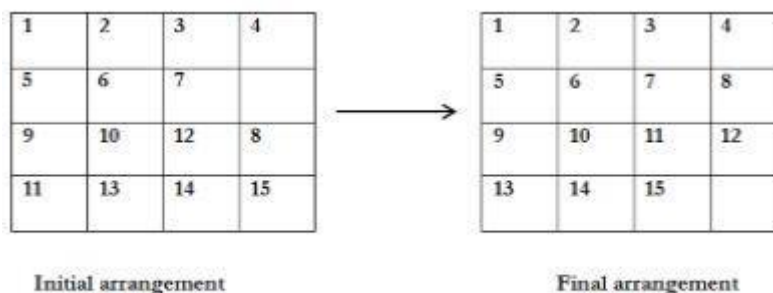
**Aim:** To study and implement 15 puzzle problem

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

**Theory:**

The 15 puzzle problem is invented by sam loyd in 1878.

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



**Figure 12**

- There is always an empty slot in the initial arrangement.
- The legal moves are the moves in which the tiles adjacent to ES are moved to either left, right, up or down.
- Each move creates a new arrangement in a tile.
- These arrangements are called as states of the puzzle.
- The initial arrangement is called as initial state and goal arrangement is called as goal state.
- The state space tree for 15 puzzle is very large because there can be 16! Different arrangements.
- A partial state space tree can be shown in figure.

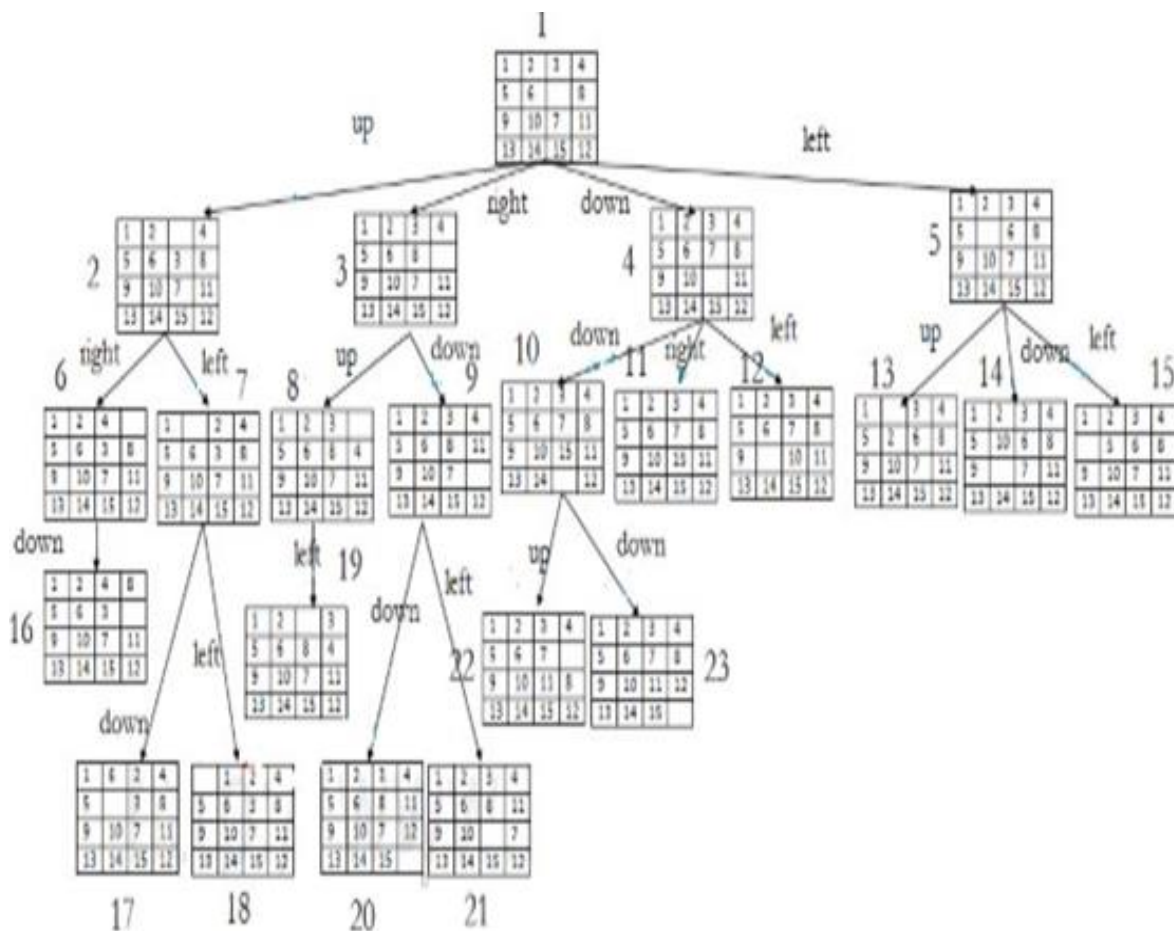


# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

- In state space tree, the nodes are numbered as per the level.
- Each next move is generated based on empty slot positions.
- Edges are label according to the direction in which the empty space moves.
- The root node becomes the E – node.
- The child node 2, 3, 4 and 5 of this E – node get generated.
- Out of which node 4 becomes an E – node. For this node the live nodes 10, 11, 12 gets generated.
- Then the node 10 becomes the E – node for which the child nodes 22 and 23 gets generated.
- Finally we get a goal state at node 23.
- We can decide which node to become an E – node based on estimation formula.

### Example:







### Implementation:

```
#include <stdio.h>
#include <stdlib.h>

#define N 4 // Size of the puzzle grid (N x N)
#define EMPTY_TILE 0

// Function to print the current state of the puzzle
void printPuzzle(int puzzle[N][N]) {
    int i,j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (puzzle[i][j] == EMPTY_TILE) {
                printf("  "); // Print empty tile
            } else {
                printf("%2d ", puzzle[i][j]);
            }
        }
        printf("\n");
    }
}

// Function to check if the puzzle is solved
int isSolved(int puzzle[N][N]) {
    int count = 1;
    int i,j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (puzzle[i][j] != count && (i != N - 1 || j != N - 1)) {
                return 0; // Puzzle is not solved
            }
            count++;
        }
    }
    return 1; // Puzzle is solved
}

// Function to move the empty tile in the puzzle
void moveTile(int puzzle[N][N], int moveX, int moveY) {
    int emptyX, emptyY, i, j;
```



```
// Find the position of the empty tile
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        if (puzzle[i][j] == EMPTY_TILE) {
            emptyX = i;
            emptyY = j;
            break;
        }
    }
}

// Swap the empty tile with the tile to be moved
puzzle[emptyX][emptyY] = puzzle[emptyX + moveX][emptyY + moveY];
puzzle[emptyX + moveX][emptyY + moveY] = EMPTY_TILE;
}

int main() {
    int puzzle[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, EMPTY_TILE}
    };

    printf("Initial Puzzle State:\n");
    printPuzzle(puzzle);

    // Example move: Move the empty tile up
    moveTile(puzzle, -1, 0);
    printf("\nPuzzle State After Move:\n");
    printPuzzle(puzzle);

    // Check if the puzzle is solved
    if (isSolved(puzzle)) {
        printf("\nPuzzle Solved!\n");
    } else {
        printf("\nPuzzle Not Solved Yet.\n");
    }

    return 0;
}
```



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

**Output:**

```
C:\TURBOC3\BIN>TC
Initial Puzzle State:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

Puzzle State After Move:
 1  2  3  4
 5  6  7  8
 9 10 11
13 14 15 12
```

**Conclusion:** The implementation of the 15 puzzle problem in C demonstrated the fundamental mechanics of puzzle manipulation and state checking. While the provided code offers a basic framework, further extensions could include implementing solving algorithms such as A\* search to find optimal solutions efficiently.



**Vidyavardhini's College of Engineering and Technology**

**Department of Artificial Intelligence & Data Science**

---

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



# Vidyavardhini's College of Engineering and Technology

## Department of Artificial Intelligence & Data Science

---

### 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  $s = 0, 1, \dots, n-m$ , 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:**

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A	A	B	A						A	A	B	A			
A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
												A	A	B	A

Pattern Found at 0, 9 and 12



### Algorithm:

#### THE NAIVE ALGORITHM

The naive 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 values of  $s$ . ( $P$ =pattern ,  $T$ =text/string ,  $s$ =shift)

**NAIVE-STRING-MATCHER( $T,P$ )**

- 1)  $n = T.length$
- 2)  $m = P.length$
- 3) **for**  $s=0$  to  $n-m$
- 4)     **if**  $P[1....m]==T[s+1....s+m]$
- 5)         **printf** "Pattern occurs with  
          shift "  $s$

### Implementation:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int match(char [], char []);
```

```
int main() {
```

```
    char a[100], b[100];
```

```
    int position;
```

```
    printf("Enter some text\n");
```

```
    gets(a);
```

```
    printf("Enter a string to find\n");
```

```
    gets(b);
```



```
position = match(a, b);
if (position != -1) {
    printf("Found at location: %d\n", position + 1);
}
else {
    printf("Not found.\n");
}
return 0;
}

int match(char text[], char pattern[]) {
    int c, d, e, text_length, pattern_length, position = -1;
    text_length = strlen(text);
    pattern_length = strlen(pattern);

    if (pattern_length > text_length) {
        return -1;
    }

    for (c = 0; c <= text_length - pattern_length; c++) {
        position = e = c;
        for (d = 0; d < pattern_length; d++) {
            if (pattern[d] == text[e]) {
                e++;
            }
            else {
                break;
            }
        }
        if (d == pattern_length) {
```



```
    return position;
}
}
return -1;
}
```

### Output:

```
Enter some text
aabbccddaabbdhgaaabbcc
Enter a string to find
aabbcc
Found at location: 1
```

```
=== Code Execution Successful ===|
```

**Conclusion:** Experiment underscores the utility of the naive string matching algorithm in efficiently locating occurrences of a pattern within a text. While straightforward in approach, its effectiveness in basic string searching tasks highlights its foundational significance in algorithmic design and serves as a benchmark for more complex pattern matching algorithms.