Department of Information Technology
BABU BANARASI DAS
Institute of Technology & Management, Lucknow
Recognized by AICTE, Govt.of India & Affiliated to Dr. A.P.J Abdul Kalam Technical University,Lucknow,AKTU College Code=054

LAB MANUAL
OF
DESIGN AND ANALYSIS OF ALGORITHM

[BCS – 553]

B.TECH, 3RD Year, Semester -V

Dr. A.P.J. Abdul Kalam Tech. University
Uttar Pradesh

2025-26

Department of Information Technology

Faculty Name:
Abhishek Dubey

Assistant Professor

Approved by Head of Department

MANUAL CONTENTS

This manual is intended for the 3rd year students of Information Technology in the subject of Design and Analysis of Algorithm Lab. This manual typically contains practical/lab sessions related Algorithms covering various aspects related to the subject of enhanced understanding.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions.

PREFACE

This practical manual will be helpful for the students of Information Technology for understanding the course from the point of view of applied aspects. 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.

VISION OF THE INSTITUTE

"To become a leading institute of providing professionally competent and socially responsive technocrats with high moral values."

MISSION OF THE INSTITUTE

- To create an eco-system for the dissemination of technical knowledge, to achieve academic excellence.
- To develop technocrats with creative skills and leadership qualities, to solve local and global challenges.
- To impart human values and ethics in students, to make them socially and Eco-friendly responsible.

.

VISION OF THE DEPARTMENT

"To transform the students to be skilled IT professionals, Innovative leaders and environmentally receptive citizens."

MISSION OF THE DEPARTMENT

M1: To develop and deliver quality academic programs in emerging & innovative fields of engineering to empower the students to meet industry standards.

M2: To build students community with ethical standards to undertake R&D in thrust areas of national & international needs.

M3: To inculcate professional behavior, positive attitude and communication skills.

.

Program Educational Objectives (PEOs) of Department

PEO1: To develop the ability among students to understand the concept of core Information Technology subjects that will facilitate understanding of any new technology.

PEO2: To embed a strong foundation in the engineering fundamentals to solve analyze and design real time engineering products.

PEO3: To give exposures to emerging edge technologies adequate training and opportunities to work as a team on multi-disciplinary projects with effective communication skills and leadership qualities.

Program Outcomes: (PO)

Graduates will be able to achieve:

PO 1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering Fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. **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.

PO 3. **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.

PO 4. 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.

PO 5. 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.

PO 6. 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.

PO 7. 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.

PO 8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. 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.

PO 11. 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.

PO 12. 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) of the Department

PSO1: Use and apply current technical concepts and practices in the core Information Technologies of Human Machine Learning, Information Management, Programming, Cloud Computing.

PSO2: Design, develop and test software systems for worldwide network of computers to provide solutions to real world problems.

Course Evaluation Scheme

Sr No.	Subject Code	Subject Name	Periods			Evaluation Scheme				Total	Credit
			L	T	P	Sessional Assessment			PE		
						CT	TA	PS			
1.	BCS553	DAA Lab	0	0	2	-	-	50	50	100	1

Course Objectives:

The teacher will explain:

1.	Able to learn brief knowledge about different algorithms
2.	Able to understand the working principle behind different algorithms.
3.	Able to implement different problem sets by using these algorithms.

Pre- requisite: Fundamentals of Computer Science and Programming, Basic Algorithms.

Course Outcomes (COs)

Course Outcomes: The students should be able to:		**Bloom's Level**
CO1	Implement algorithm to solve problems by iterative approach.	K2
CO2	Implement algorithm to solve problems by divide and conquer approach.	K3
CO3	Implement algorithm to solve problems by Greedy algorithm approach.	K4
CO4	Implement algorithm to solve problems by Dynamic programming, backtracking, branch and bound approach.	K5
CO5	Implement algorithm to solve problems by branch and bound approach.	K4

CO-PO-PSO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	1	3	2	2	1	-	-	-	-	-	-	3	1	1
CO2	2	3	2	3	2	-	-	-	-	-	-	2	1	2
CO3	3	2	2	1	2	-	-	-	-	-	-	2	1	2
CO4	2	2	2	1	2	-	-	-	-	-	-	2	2	2
CO5	2	2	2	2	1	-	-	-	-	-	-	2	2	2

The extent of mapping is as follows: 1 for low, 2 for moderate, 3 for high &"-" for No correlation between CO & PO.

List of Programs with enhancement of Faculty

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

Beyond Syllabus:

11	Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate how the divide and- conquer method works along with its time complexity analysis: worst case, average case and best case.
12	Design and implement to find a subset of a given set S = {Sl, S2,. ... ,Sn} of n positive integers whose SUM is equal to agiven positive integer d. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display asuitable message, if the given problem instance doesn't have a solution.

Department of Information Technology

DAA LAB (BCS – 553)

List of Experiments

S. No.	Name of the Experiment	Corresponding	Blooms Level
1.	Program for Recursive Binary & Linear Search using python.	1	1
2.	Program for Heap Sort using python.	1	1
3.	Program for Merge Sort using python.	2	2
4.	Program for Selection Sort using python.	2	2
5.	Program for Insertion Sort using python.	3	3
6.	Program for Quick Sort using python.	3	3
7.	Knapsack Problem using Greedy Solution.	4	4
8.	Perform Travelling Salesman Problem using python.	4	4
9.	Find Minimum Spanning Tree using Kruskal's Algorithm.	5	5
10.	Implement N Queen Problem using Backtracking.	5	5

Beyond Syllabus

11.	Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program forvaried values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus non graph sheet. Theelements can be read from a file or can be generated using the random number generator. Demonstrate how the divideand- conquer method works along with	4	4

	its time complexity analysis: worst case, average case and best case.		
12.	Design and implement to find a subset of a given set S = {Sl, S2,. ... ,Sn} of n positive integers whose SUM is equal to agiven positive integer d. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display asuitable message, if the given problem instance doesn't have a solution.	4	4

Subject Coordinator
Abhishek Dubey

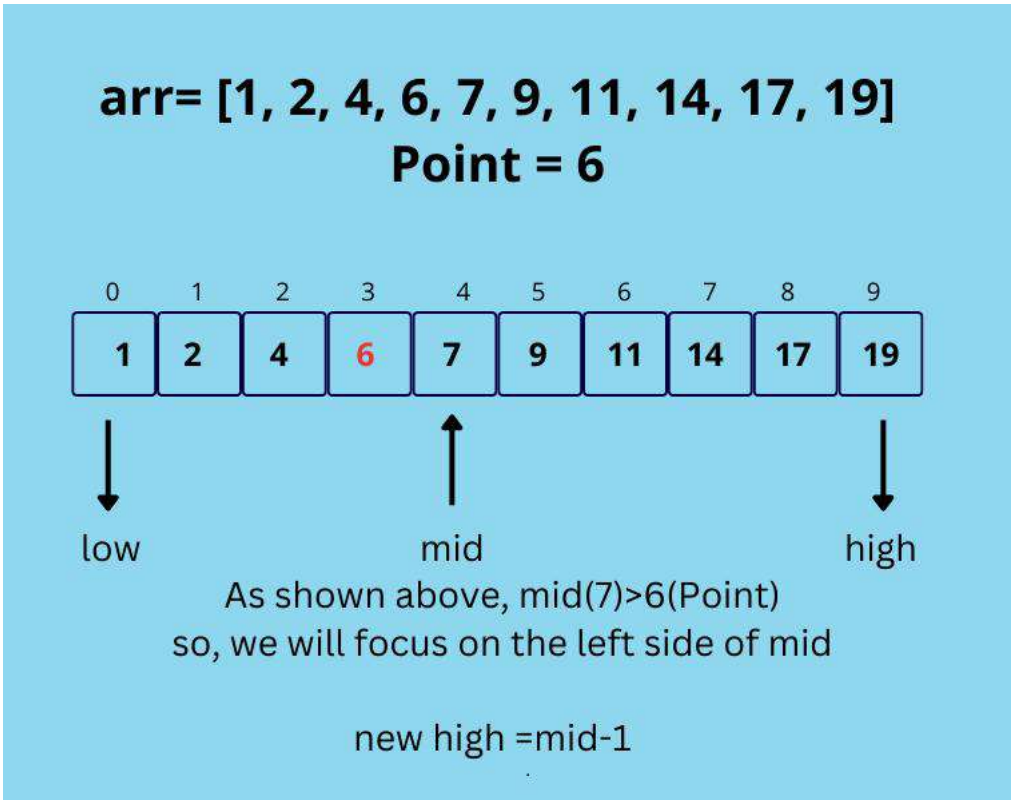
Experiment-1

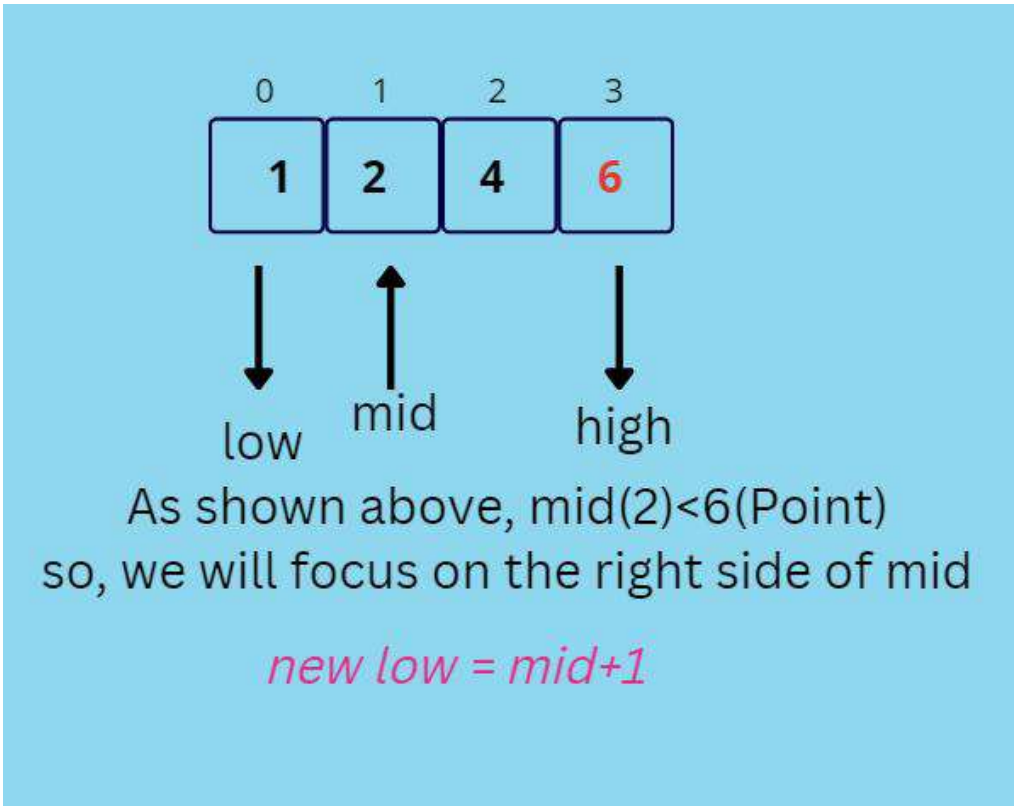
Problem:
Program for Recursive Binary & Linear Search using python.

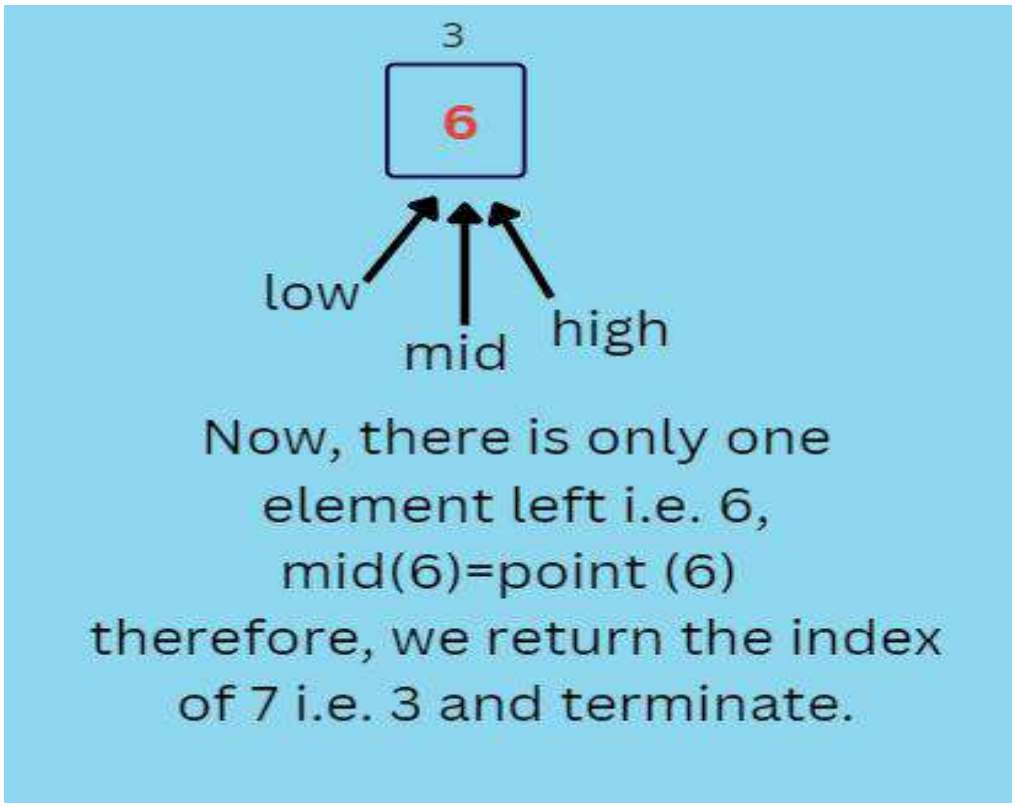
Working:

Recursive Binary Search Algorithm:

1. Find the element at arr[size/2], which will be the array's midpoint. The array is split halfway, with the lower half consisting of items 0 to midpoint -1 and the top half consisting of elements midpoint to size -1.
2. Compare the *key* to *arr[midpoint]* by calling the user function.
3. If the key is a match, return *arr[midpoint];*
4. Otherwise return NULL, indicating that there is no match if the array has only one element.
5. Search the lower half of the array by repeatedly executing search if the key is less than the value taken from arr[midpoint].
6. Call search recursively to search the upper half of the array.




0
1
2
3
1
2
4
6
low
mid
high
As shown above, mid(2)<6(Point)
so, we will focus on the right side of mid
new low = mid+1


3
6
low
mid
high
Now, there is only one element left i.e. 6, mid(6)=point (6) therefore, we return the index of 7 i.e. 3 and terminate.

Code:

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = low + (high - low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        return -1

arr = [2, 3, 4, 10, 40]
x = 10

result = binary_search(arr, 0, len(arr) - 1, x)

if result != -1:
    print("Element is present at index", result)
else:
    print("Element is not present in array")
```

Output:

Element is present at index 3

Experiment-2

Problem: Program for Heap Sort using python.

Working:

Heap Sort Algorithm:
Heap Sort works in two main phases:

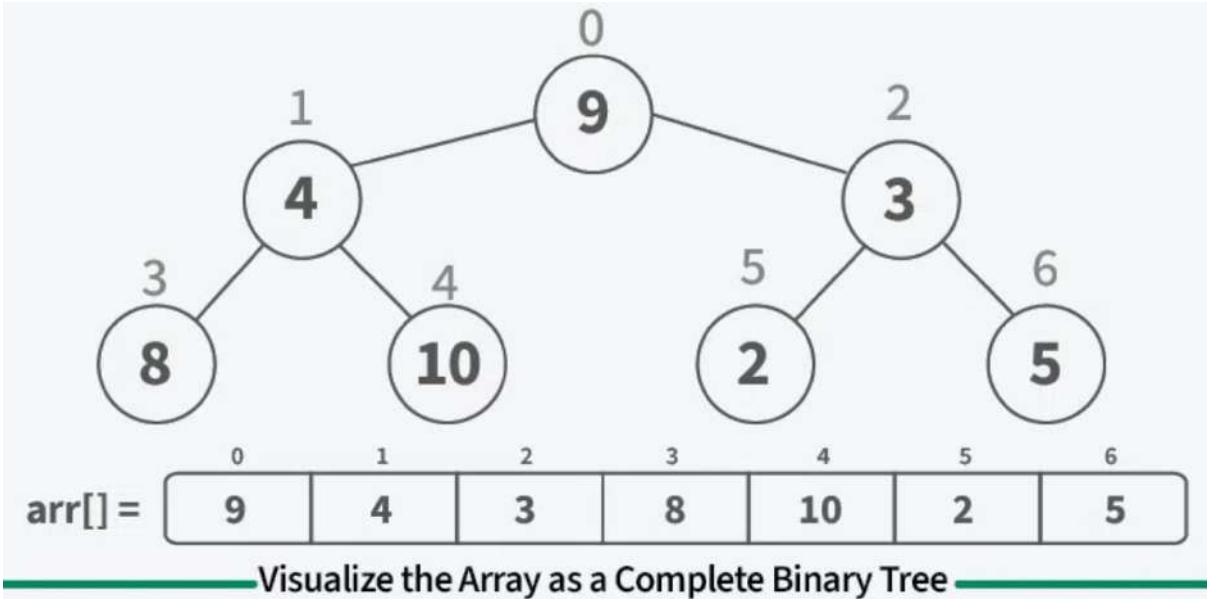
1. Build a Max Heap from the input data.
2. Extract elements from the heap one by one, placing the maximum element at the end each time.

Algorithm Steps:

1. Rearrange array elements to form a Max Heap.
2. Repeat until the heap has only one element:
3. Swap the root (maximum element) with the last element of the heap.
4. Reduce heap size by 1.
5. Heapify the root to restore heap property.
6. The array is now sorted in ascending order.

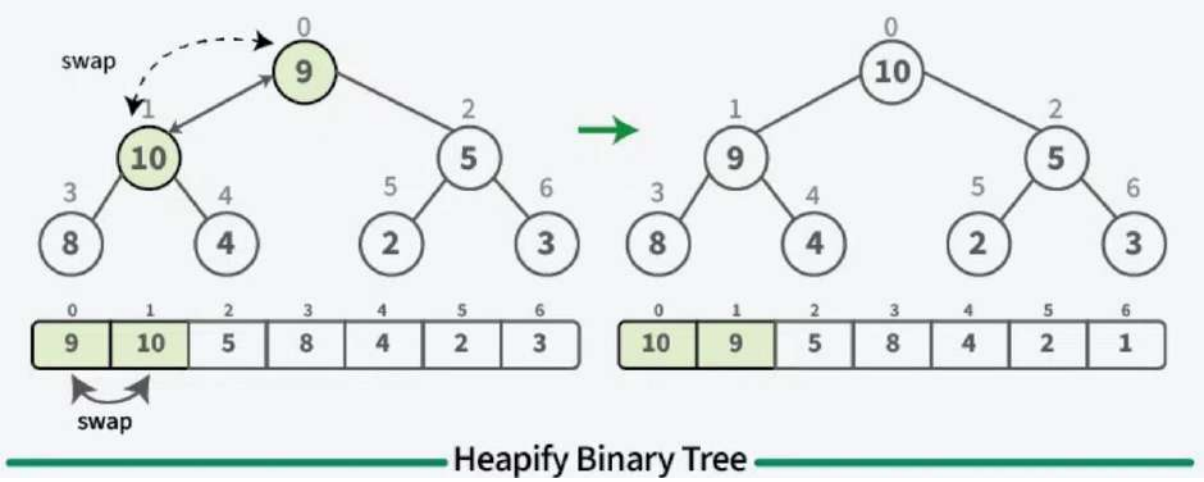
How Heap Sort Works:
Step 1: Represent the Array as a Binary Tree
For an array of size n,

- Root is at index 0
- Left child of node **i: 2*i + 1**
- Right child of node **i: 2*i + 2**



Step 2: Build a Max Heap

Convert the array into a max heap where each parent is larger than its children.



Code:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap max to end
        heapify(arr, i, 0)

arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
print("Sorted array is:", arr)
```

Output:

('Sorted array is:', [5, 6, 7, 11, 12, 13])

Experiment-3

Problem:
Program for Merge Sort using python.

Working:

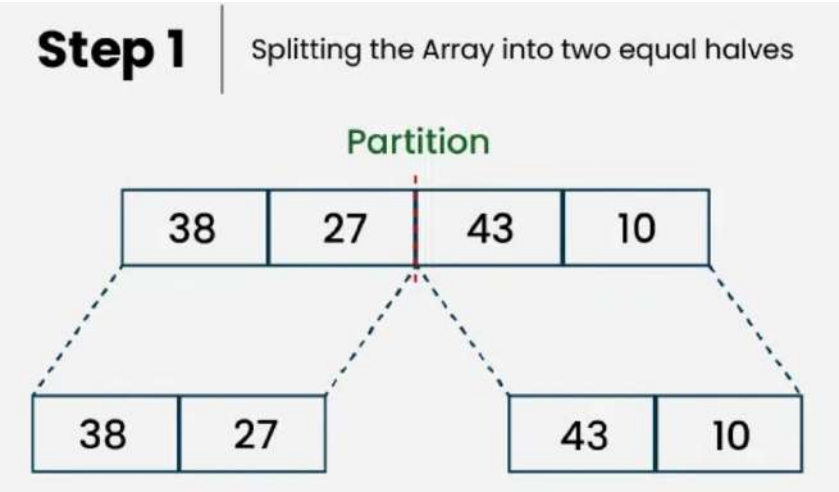
Merge Sort in Python:

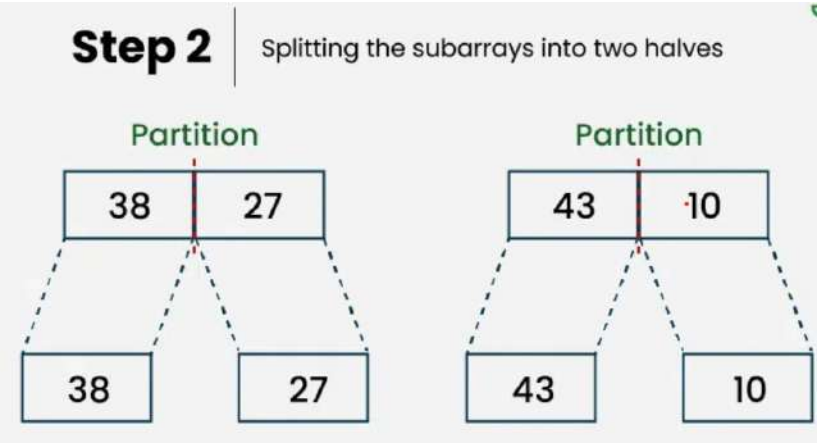
Merge Sort is one of the most efficient and stable sorting algorithms based on the Divide and Conquer technique. It divides an input array into two halves, recursively sorts them, and then merges the two sorted halves using a function called merge().

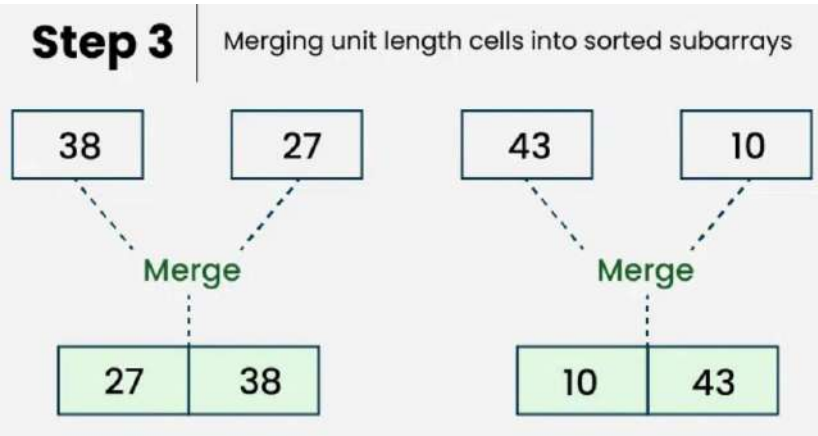
The key function, merge(arr, l, m, r), assumes that both halves **arr[l..m]** (left half) and **arr[m+1..r]** (right half) are already sorted and merges them into one sorted array.

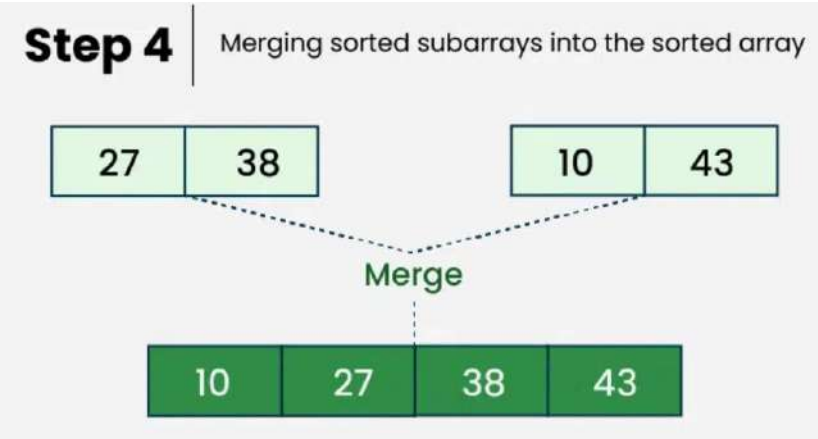
How Merge Sort Works:
Here's the step-by-step breakdown:

1. **Divide**: Recursively split the array into two halves until each subarray has only one element.
2. **Conquer**: Sort each subarray individually (they're already sorted if they contain one element).
3. **Merge**: Combine the sorted subarrays into a single sorted array in the correct order.




Step 2
Splitting the subarrays into two halves
Partition
Partition
38
27
43
10
38
27
43
10


Step 3
Merging unit length cells into sorted subarrays
38
27
43
10
Merge
Merge
27
38
10
43


Step 4
Merging sorted subarrays into the sorted array
27
38
10
43
Merge
10
27
38
43

Code:

```
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid
    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0
    j = 0
    k = left

    # Merge the temp arrays back
    # into arr[left..right]
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[],
    # if there are any
    while j < n2:
        arr[k] = R[j]
```

```
        j += 1
        k += 1

def mergeSort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)

# Driver code
if __name__ == "__main__":
    arr = [38, 27, 43, 10]

    mergeSort(arr, 0, len(arr) - 1)
    for i in arr:
        print(i, end=" ")
    print()
```

Output:

10 27 38 43

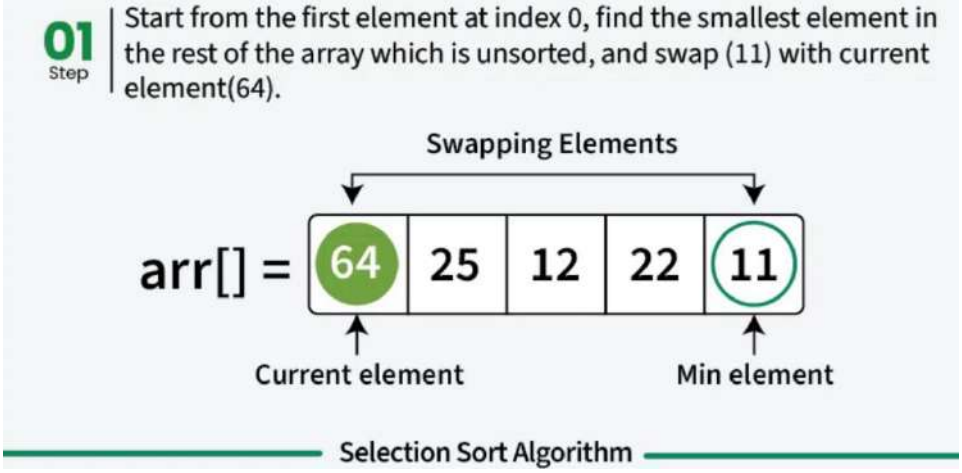
Experiment-4

Problem: Program for Selection Sort using python.

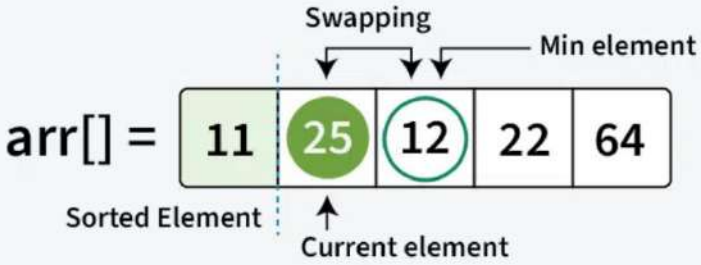
Working:

Selection Sort is a comparison-based sorting algorithm. It sorts by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.

1. Find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

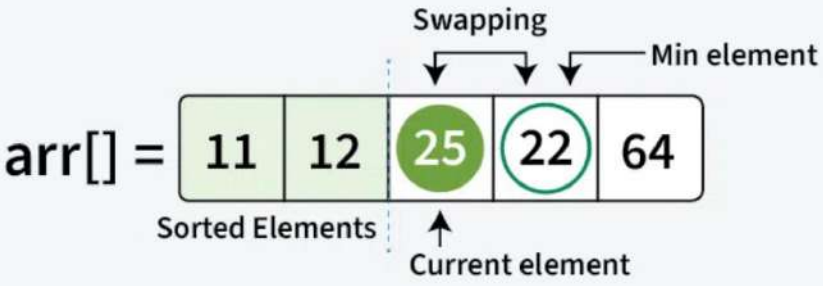


02 Step | Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



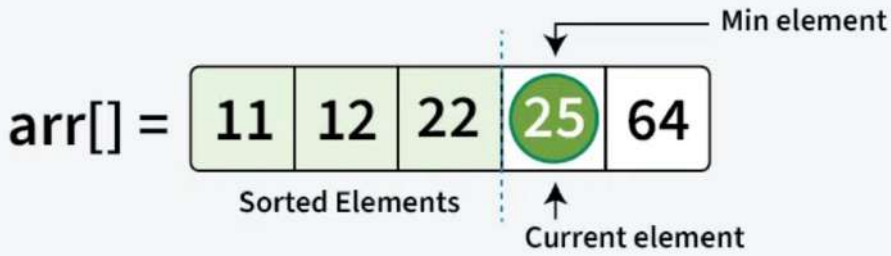
Selection Sort Algorithm

03 Step | Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



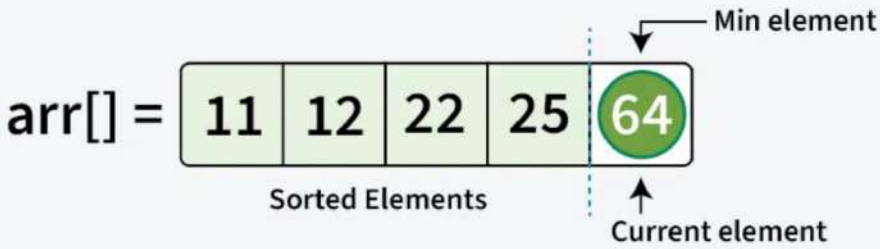
Selection Sort Algorithm

04 Step | Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).

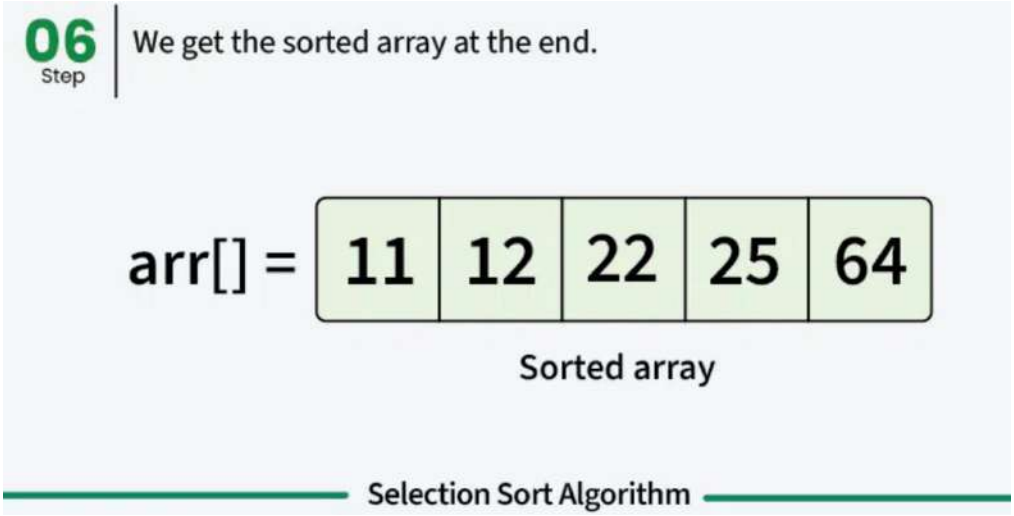


Selection Sort Algorithm

05 Step | Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).



Selection Sort Algorithm



Code:

```
def selection sort(arr):
    n = len(arr)
    for i in range(n - 1):

        min_idx = i

        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:

                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def print_array(arr):
    for val in arr:
        print(val, end=" ")
    print()

if __name__ == "__main__":
    arr = [64, 25, 12, 22, 11]
```

```
print("Original array: ", end="")
print_array(arr)

selection_sort(arr)

print("Sorted array: ", end="")
print_array(arr)
```

Output:
Original vector: 64 25 12 22 11
Sorted vector: 11 12 22 25 64

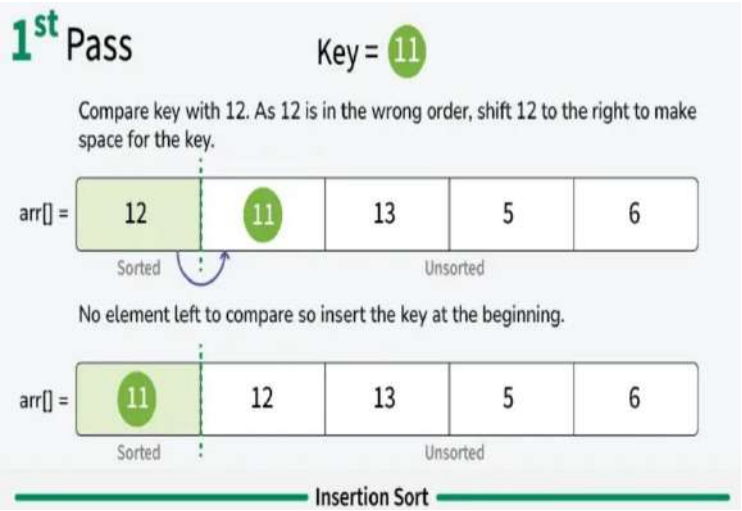
Experiment: 5

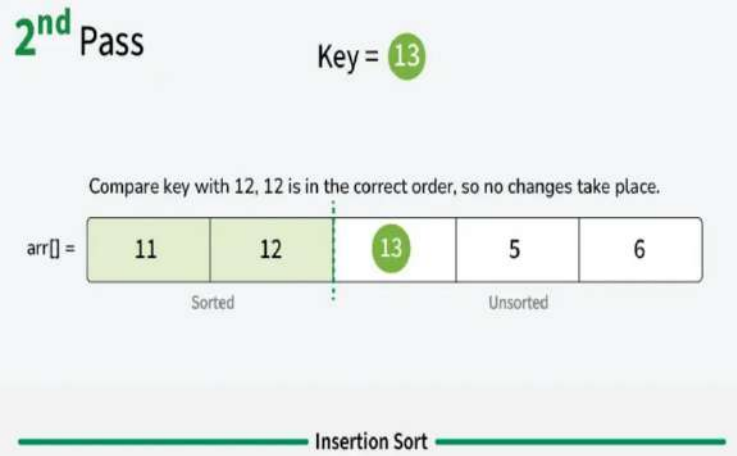
Problem: Program for Insertion Sort using python.

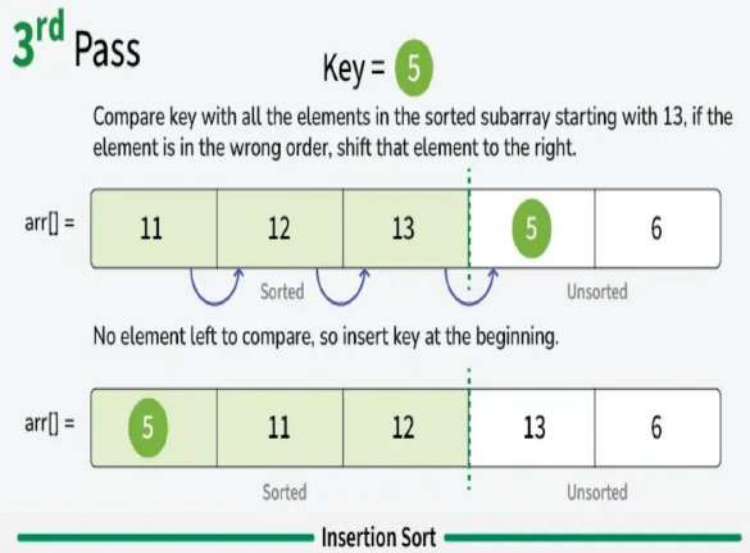
Working:

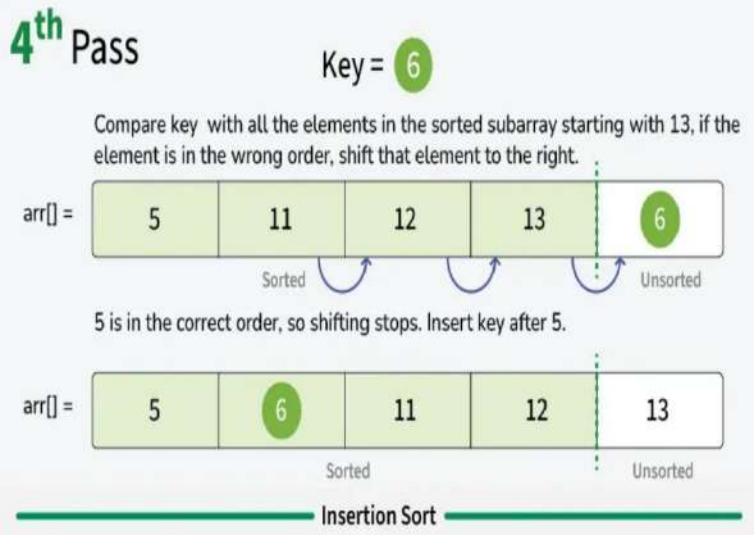
Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

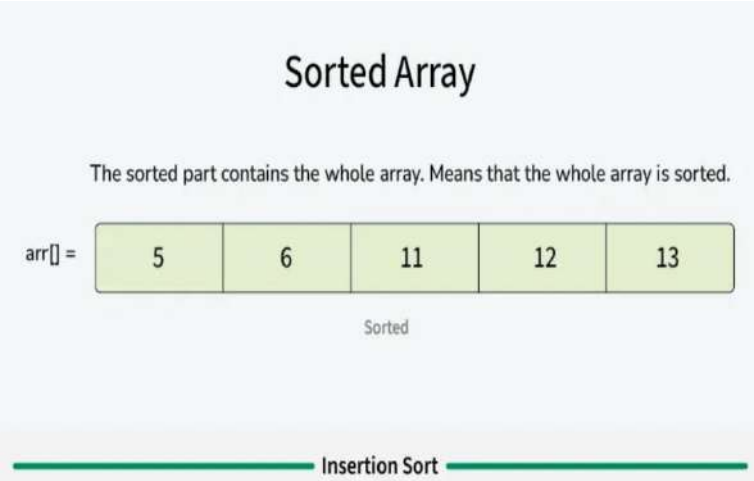
- Start with the second element as the first element is assumed to be sorted.
- Compare the second element with the first if the second is smaller then swap them.
- Move to the third element, compare it with the first two, and put it in its correct position.
- Repeat until the entire array is sorted.




2nd Pass
Key = 13
Compare key with 12, 12 is in the correct order, so no changes take place.
arr[] =
11
12
13
5
6
Sorted
Unsorted
Insertion Sort


3rd Pass
Key = 5
Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.
arr[] =
11
12
13
5
6
Sorted
Unsorted
No element left to compare, so insert key at the beginning.
arr[] =
5
11
12
13
6
Sorted
Unsorted
Insertion Sort





Code:

```
def insertionSort(arr):
    for i in range(1, len(arr)):
```

```
        key = arr[i]
        j = i – 1

while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def printArray(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6]
    insertionSort(arr)
    printArray(arr)
```

Output:

5 6 11 12 13

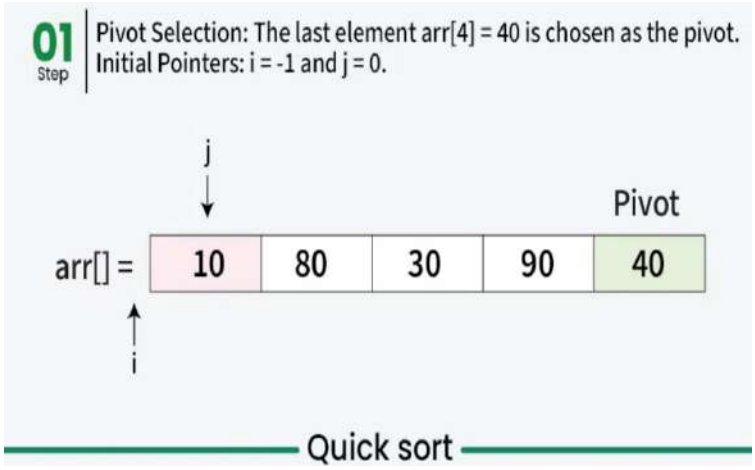
Experiment: 6

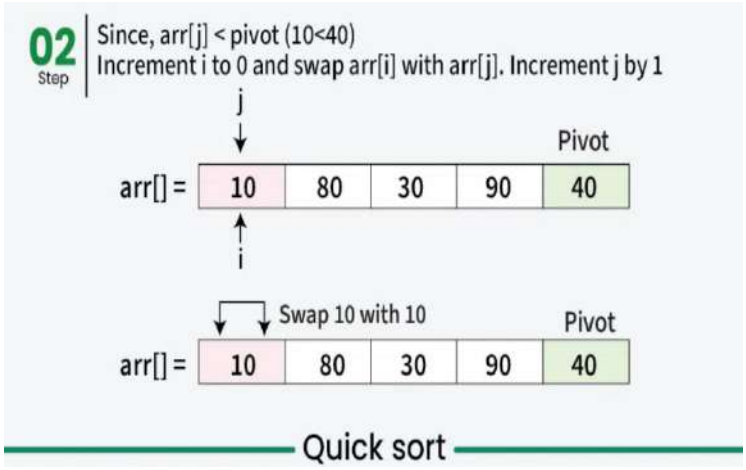
Problem: Program for Quick Sort using python.

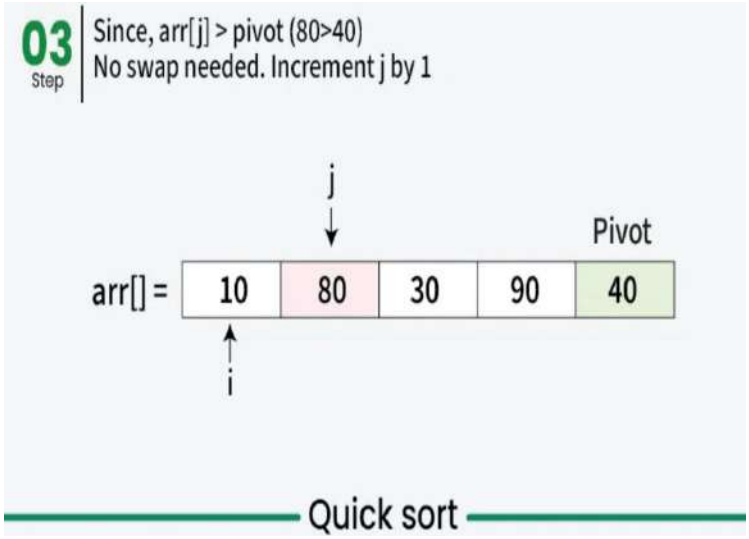
Working:

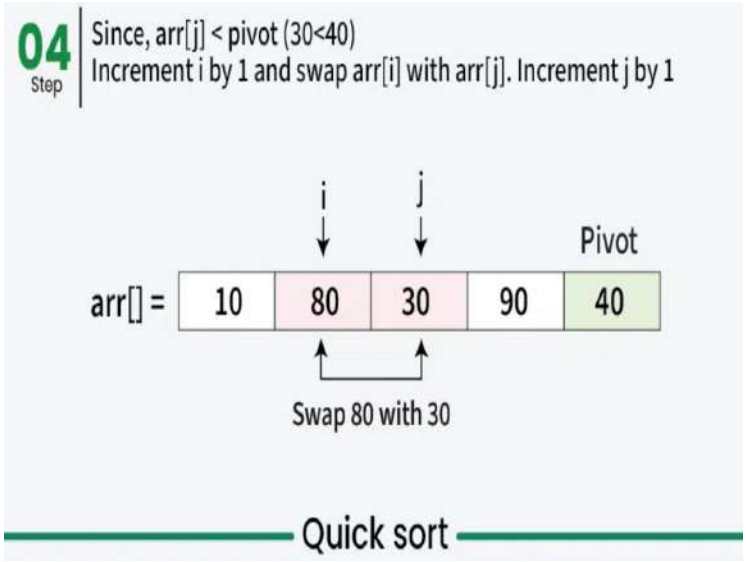
Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. .
There are mainly three steps in the algorithm:

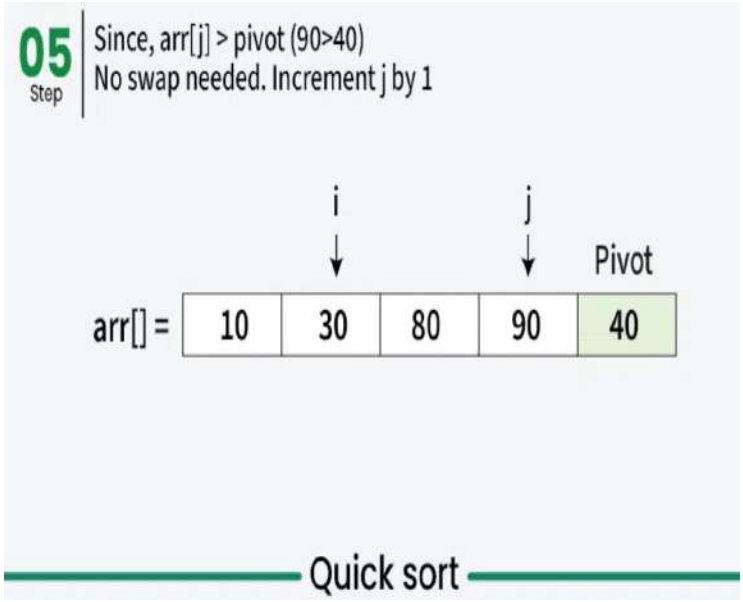
1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays.
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

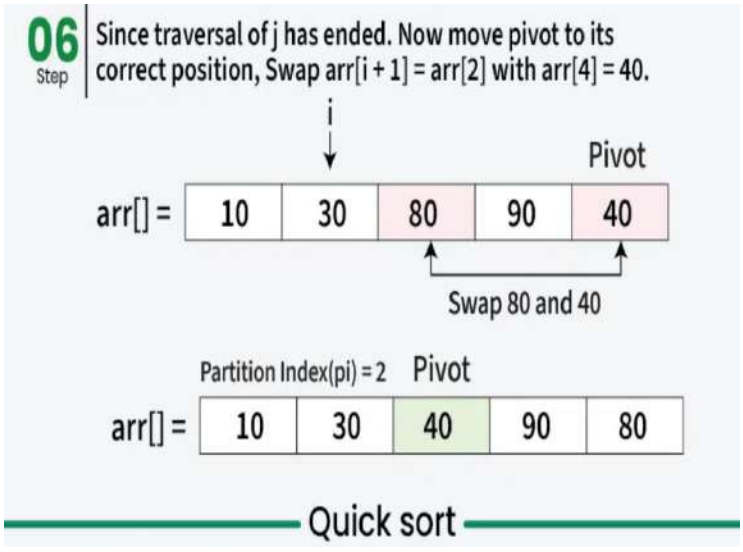





03
Step
Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1
j
Pivot
arr[] =
10
80
30
90
40
i
Quick sort


04
Step
Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1
i
j
Pivot
arr[] =
10
80
30
90
40
Swap 80 with 30
Quick sort


05
Step
Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1
i
j
Pivot
arr[] =
10
30
80
90
40
Quick sort


06
Step
Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.
i
Pivot
arr[] =
10
30
80
90
40
Swap 80 and 40
Partition Index(pi) = 2
Pivot
arr[] =
10
30
40
90
80
Quick sort

Code:

```
def partition (arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            swap(arr, i, j)

swap(arr, i + 1, high)
   return i + 1

def swap(arr, i, j):
   arr[i], arr[j] = arr[j], arr[i]

def quickSort(arr, low, high):
   if low < high:

pi = partition(arr, low, high)

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

if __name__ == "__main__":
   arr = [10, 7, 8, 9, 1, 5]
   n = len(arr)


 quickSort(arr, 0, n - 1)

   for val in arr:
      print(val, end=" ")
```

Output:

1 5 7 8 9 10

Experiment: 7

Problem: Knapsack Problem using Greedy Solution.

Working:

Knapsack Problem:
Firstly, we have given a knapsack of the maximum capacity of m kg and n items with their weight and profit. Fill in the knapsack with a subset of items such that the selected weight is less than or equal to the capacity of the knapsack and the profit of items is maximum.

Algorithm of solving Knapsack Problem using Greedy Method:
- Assume a knapsack of capacity M and n items having profit p_i and weight w_i
- Sort items by profit/weight ratio: p_i/w_i
- Consider items in order of decreasing ratio
- Take as much of each item as possible.
- Traverse this sorted list as:

```
If (wi ≤ M)
        {
          Xi=1;
          M=M-wi;
        }
        else if (wi > M && M>0)
        {
          Xi=M/wi;
          M=0;
        }
        else
        {
         Xi=0;
        }
```

Example:
Let us consider that the capacity of the knapsack M=60 and the list of provided items are shown in the following table:

ITEM	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio(**pi/wi**)	7	10	6	5

However, the provided items are not sorted based on (pi/wi). After sorting, the items are shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio(**pi/wi**)	10	7	6	5
Xi	1	1	10/20	0

Solution:
Afterward, sorting all the items according to (pi/wi), first item B is chosen as the weight of B is less than the capacity of the knapsack. Next, item A is chosen, as the available capacity of the knapsack is greater than the weight of A. Then, C is chosen as the next item.

However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of C.

Hence, fraction of C (i.e.(60-50)/20)is chosen.

Chiefly, the capacity of the knapsack is equal to the selected items. Hence, no more items can be selected.

So, the total weight of the selected items is 10+40+20*(10/20)=60

Also, the total profit is 100+280+120*(10/20)=380+60=440

Hence, this is the optimal solution. Moreover, we cannot gain more profit by selecting any different combination of items.

Code:

```
function compare (a, b) {
   let a1 = (1.0 * a[0]) / a[1];
   let b1 = (1.0 * b[0]) / b[1];
   return b1 - a1;
}

function fractional Knapsack(val, wt, capacity) {
   let n = val.length;

   // Create array to store value and weight
   // items[i][0] = value, items[i][1] = weight
   let items = [];
   for (let i = 0; i < n; i++) {
      items.push([val[i], wt[i]]);
   }

   // Sort items based on value-to-weight ratio in descending order
```

```
    items.sort(compare);

    let res = 0.0;
    let currentCapacity = capacity;

    // Process items in sorted order
    for (let i = 0; i < n; i++) {

        // If we can take the entire item
        if (items[i][1] <= currentCapacity) {
            res += items[i][0];
            currentCapacity -= items[i][1];
        }

        // Otherwise take a fraction of the item
        else {
            res += (1.0 * items[i][0] / items[i][1]) * currentCapacity;

            // Knapsack is full
            break;
        }
    }

    return res;
}
// Driver Code
let val = [60, 100, 120];
let wt = [10, 20, 30];
let capacity = 50;
```

Output:

240

Experiment: 8

Problem: Perform Travelling Salesman Problem using python.

Working:

The Traveling Salesman Problem (TSP) is a classic algorithmic problem in the fields of computer science and operations research. It involves finding the shortest possible route that visits a set of cities and returns to the origin city. The challenge of TSP lies in its NP-hard nature, meaning that as the number of cities increases, the problem becomes exponentially more complex to solve optimally.

Implementing Traveling Salesman Problem (TSP) in Python:

Let's implement a simple solution using dynamic programming (Held-Karp algorithm) in Python. This method involves breaking the problem into smaller subproblems and solving each subproblem only once, storing the results to avoid redundant calculations.

Step-by-Step Implementation:

1. **Define the Problem:** Represent the cities and the distances between them using a distance matrix.
2. **Initialize the Dynamic Programming Table:** Create a table to store the minimum costs of visiting subsets of cities.
3. **Iterate Through Subsets:** Compute the minimum costs for visiting subsets of cities and update the table.
4. **Compute the Optimal Path:** Reconstruct the optimal path from the dynamic programming table.

Code:

```
# Python program to find the shortest possible route
# that visits every city exactly once and returns to
# the starting point

from itertools import permutations

def tsp(cost):

    # Number of nodes
    numNodes = len(cost)
    nodes = list(range(1, numNodes))

    minCost = float('inf')
```

```
    # Generate all permutations of the
    # remaining nodes
    for perm in permutations(nodes):
        currCost = 0
        currNode = 0

        # Calculate the cost of the current permutation
        for node in perm:
            currCost += cost[currNode][node]
            currNode = node

        # Add the cost to return to the starting node
        currCost += cost[currNode][0]

        # Update the minimum cost if the current cost
        # is lower
        minCost = min(minCost, currCost)

    return minCost


if __name__ == "__main__":

    cost = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    res = tsp(cost)
    print(res)
```

Output:

80

Experiment: 9

Problem: Find Minimum Spanning Tree using Kruskal's Algorithm.

Working:

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, and undirected graph is a spanning tree (no cycles and connects all vertices) that has minimum weight. The weight of a spanning tree is the sum of all edges in the tree.

Below are the steps for finding MST using Kruskal's algorithm:

- Sort all the edges in a non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it. It uses the Disjoint Sets to detect cycles.
- Repeat step 2 until there are (V-1) edges in the spanning tree.

It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Time Complexity: O(E * log E) or O(E * log V)

- Sorting of edges takes O(E*logE) time.
- After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.
- So overall complexity is O(E*logE + E*logV) time.
- The value of E can be at most O(V2), so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)

Code:

```
from functools import cmp_to_key

def comparator(a,b):
    return a[2] - b[2];

def kruskals_mst(V, edges):
```

```
    # Sort all edges
    edges = sorted(edges,key=cmp_to_key(comparator))

    # Traverse edges in sorted order
    dsu = DSU(V)
    cost = 0
    count = 0
    for x, y, w in edges:

        # Make sure that there is no cycle
        if dsu.find(x) != dsu.find(y):
            dsu.union(x, y)
            cost += w
            count += 1
            if count == V - 1:
                break
    return cost

# Disjoint set data structure
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, x, y):
        s1 = self.find(x)
        s2 = self.find(y)
        if s1 != s2:
```

```
        if self.rank[s1] < self.rank[s2]:
            self.parent[s1] = s2
        elif self.rank[s1] > self.rank[s2]:
            self.parent[s2] = s1
        else:
            self.parent[s2] = s1
            self.rank[s1] += 1


if __name__ == '__main__':

    # An edge contains, weight, source and destination
    edges = [[0, 1, 10], [1, 3, 15], [2, 3, 4], [2, 0, 6], [0, 3, 5]]
    print(kruskals_mst(4, edges))
```

Output:

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

Experiment: 10

Problem: Implement N Queen Problem using Backtracking.

Working:

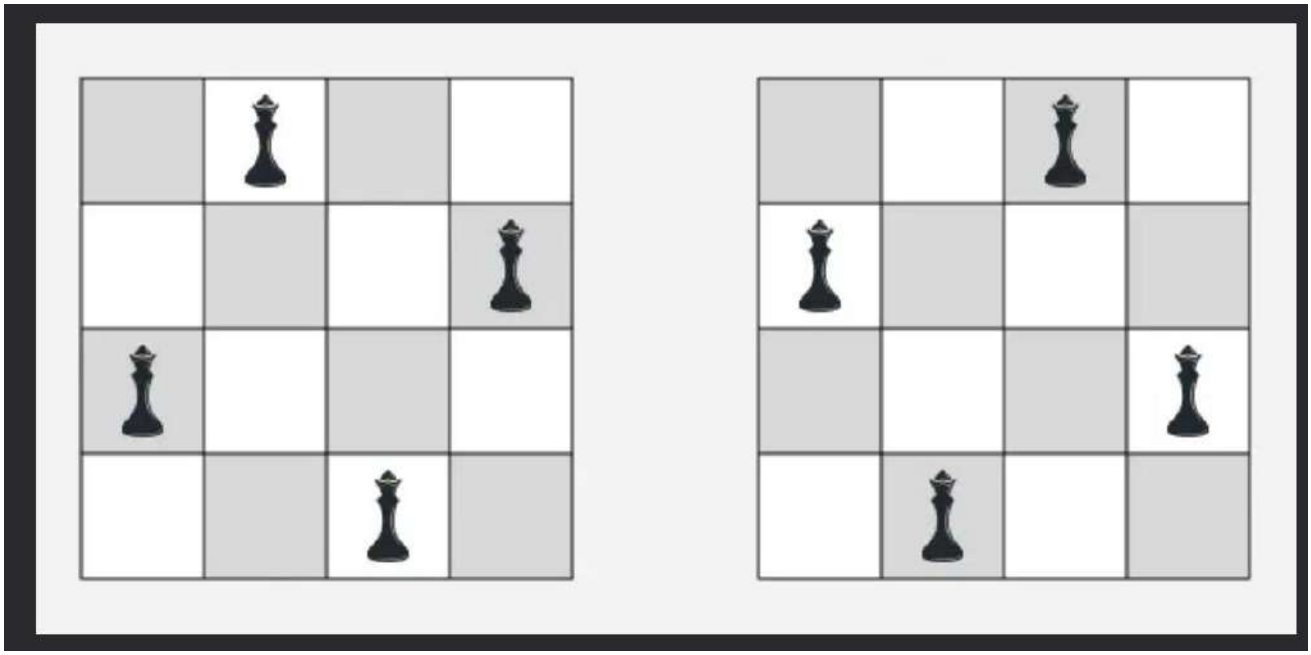
Given an integer n, place n queens on an n × n chessboard such that no two queens attack each other. A queen can attack another queen if they are placed in the same row, the same column, or on the same diagonal.

Find all possible distinct arrangements of the queens on the board that satisfy these conditions.

The output should be an array of solutions, where each solution is represented as an array of integers of size n, and the i-th integer denotes the column position of the queen in the i-th row. If no solution exists, return an empty array.

Examples:

Input:n=4
Output: [[2, 4, 1, 3], [3, 1, 4, 2]]
Explanation: Below is Solution for 4 queen problem.



Input: n = 3
Output: []
Explanation: There are no possible solutions for n = 3

Code:

```
def isSafe(mat, row, col):
    n = len(mat)

    # Check this col on upper side
    for i in range(row):
        if mat[i][col]:
            return 0

    # Check upper diagonal on left side
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if mat[i][j]:
            return 0
        i -= 1
        j -= 1

    # Check upper diagonal on right side
    i, j = row - 1, col + 1
    while i >= 0 and j < n:
        if mat[i][j]:
            return 0
        i -= 1
        j += 1

    return 1

# Recursive function to place queens
def placeQueens(row, mat, result):
    n = len(mat)

    # base case: If all queens are placed
    if row == n:

        # store current solution
        ans = []
        for i in range(n):
```

```
            for j in range(n):
                if mat[i][j]:
                    ans.append(j + 1)
        result.append(ans)
        return

    # Consider the row and try placing
    # queen in all columns one by one
    for i in range(n):

        # Check if the queen can be placed
        if isSafe(mat, row, i):
            mat[row][i] = 1
            placeQueens(row + 1, mat, result)

            # backtrack
            mat[row][i] = 0

# Function to find all solutions
def nQueen(n):

    # Initialize the board
    mat = [[0] * n for _ in range(n)]
    result = []
    # Place queens
    placeQueens(0, mat, result)
    return result
if __name__ == "__main__":
    n = 4
    result = nQueen(n)
    for ans in result:
        print(" ".join(map(str, ans)))
```

Output:

2 4 1 3
3 1 4 2

Experiment: 11

Problem:

Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate how the divide and- conquer method works along with its time complexity analysis: worst case, average case and best case.

Code:

```
import time
import random
import sys

# Increase recursion depth limit for large inputs
sys.setrecursionlimit(2000000)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Conquer (Recursive calls)
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Combine (Merge)
    return merge(left_half, right_half)

def merge(left, right):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged_list.append(left[left_index])
            left_index += 1
        else:
            merged_list.append(right[right_index])
            right_index += 1

    # Append remaining elements
    while left_index < len(left):
        merged_list.append(left[left_index])
        left_index += 1
    while right_index < len(right):
        merged_list.append(right[right_index])
```

```
        right_index += 1

    return merged_list

# Function to run experiment and record time
def run_experiment(n_values):
    times = {}
    for n in n_values:
        # Generate n random integer elements
        data = [random.randint(0, 1000000) for _ in range(n)]

        start_time = time.time()
        merge_sort(data)
        end_time = time.time()

        time_taken = end_time - start_time
        times[n] = time_taken
        print(f"n = {n}: Time taken = {time_taken:.4f} seconds")
    return times

# Experiment with varied values of n > 5000
n_values_to_test = [5000, 10000, 50000, 100000]
recorded_times = run_experiment(n_values_to_test)
```

Experiment No 12

Problem:

Design and implement to find a subset of a given set S = {Sl, S2,. ... ,Sn} of n positive integers whose SUM is equal to agiven positive integer d. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display asuitable message, if the given problem instance doesn't have a solution.

Working:

Algorithm Design:

The problem is a classic subset sum problem, which can be efficiently solved using a backtracking algorithm. The general approach involves exploring a state-space tree where each node represents a partial solution, and we decide whether to include or exclude each element of the set.

The algorithm's logic is as follows:

- Sorting the input set can help with optimizations, such as pruning the search tree if the current sum plus the remaining elements is less than the target sum, or if adding the current element exceeds the target.
- A recursive function is used to explore the solution space.
- The function maintains the current index, the current sum, and the current subset of elements.
- Base cases:
  - If the current sum equals the target d, a valid subset is found and recorded.
  - If the current sum exceeds d or all elements have been considered, the recursion stops (backtracks).
- Recursive steps:
  - For each element, the algorithm branches into two possibilities: including the element in the current subset or excluding it.
  - By systematically exploring all branches, all possible valid subsets are identified.
- If no subsets are found after exploring all possibilities, a message indicating no solution exists is displayed.

Code:

```
def find_subsets(s_set, d_target):
    solutions = []
    # Sort the set to handle duplicates and optimize pruning
    s = sorted(list(s_set))
    n = len(s)

    def backtrack_v2(index, current_sum, current_subset):
        if current_sum == d_target:
            solutions.append(list(current_subset))
            return

        if current_sum > d_target or index == n:
            return

        for i in range(index, n):
            # Skip duplicates to avoid identical subsets
            if i > index and s[i] == s[i-1]:
                continue
```

```
        # Optimization: since it's sorted, remaining elements will also exceed d_target
        if current_sum + s[i] > d_target:
            continue

        current_subset.append(s[i])
        backtrack_v2(i + 1, current_sum + s[i], current_subset)
        current_subset.pop() # Backtrack: remove the last added element

    backtrack_v2(0, 0, [])
    return solutions

S_set = {1, 2, 5, 6, 8}
D_target = 9
result = find_subsets(S_set, D_target)

if result:
    print(f"Solutions for S={S_set}, d={D_target}: {result}")
else:
    print(f"For S={S_set} and d={D_target}, no solution exists.")

# Test case with no solution:
S_set_no = {1, 2, 3}
D_target_no = 10
result_no = find_subsets(S_set_no, D_target_no)

if result_no:
    print(f"Solutions for S={S_set_no}, d={D_target_no}: {result_no}")
else:
    print(f"For S={S_set_no} and d={D_target_no}, no solution exists.")
```

Output:

S= {1, 2, 5, 6, 8}, d=9: [[1, 2, 6], [1, 8]]