PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Engineering & Technology

Department of Information and Communication Engineering

Course name : Data Structure and Algorithm Sessional

Course Code : ICE-2202

Lab Report On Data Structure and Algorithm

Submitted By:

Name: Arifur Rahman
Roll:220612
Session:2021-22
2nd Year 2nd Semester
Department of ICE,PUST

Submitted To:

Prof. Md. Anwar Hossain
Chairman
Department of Information and Communication Engineering,
Pabna University of Science and Technology, Pabna

Index

Experiment no : 1

Experiment name : Write a program to sort a linear array using bubble sort algorithm.

Title : C program to sort a linear array using bubble sort algorithm.

Theory :

- ❖ Sorting is a fundamental problem in computer science where we arrange elements in a specific order. In this case, we want to sort an array of numbers in ascending order using the Bubble sort technique.The goal of the program is to sort a list of integers input by the user. The sorting is done using the Bubble Sort algorithm, which repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed, indicating that the list is sorted.

We can breakdown the logic of program in 3 phases:

1. Input Phase:

   *The program first prompts the user to enter the number of elements they wish to sort.

   *It then allocates an array of that size and asks for the individual elements.

2. Sorting Phase:

   The program implements Bubble Sort through nested loops:

   *The outer loop runs through each element of the array.

   *The inner loop compares adjacent elements and swaps them if they are out of order.

   *After each pass of the outer loop, it prints the current state of the array.

3. Output Phase:

   *Once sorting is complete, the program outputs the sorted array

Algorithm :

Step 1: Input the Array:

1. Start the program.
2. Prompt the user to enter the number of elements (n) in the array.
3. Read the value of n.
4. Declare an array of size n.
5. Prompt the user to enter n elements of the array.
6. Read and store the elements in the array.

Step 2: Apply Bubble Sort Algorithm:

1. Loop through the array n-1 times:
    1. Outer Loop (i = 0 to n-2):
        1. Represents each pass of the sorting process.
2. In each pass, compare adjacent elements and swap them if needed:
    1. Inner Loop (j = 0 to n-i-2):
        1. Compare arr[j] with arr[j+1].
        2. Print the elements being compared.
        3. If arr[j] > arr[j+1], swap them.
        4. Print the swapped elements.
        5. Print the current state of the array after each swap.
3. After each pass, print the array state.

Step 3: Output the Sorted Array:

1. Print the sorted array in ascending order.
2. End the program.

Program :

```
#include <stdio.h>

int main() {
```

```
int n;

// Input the size of the array
printf("Enter the number of elements in the array: ");
scanf("%d", &n);

int arr[n];

// Input the elements of the array
printf("Enter the elements of the array: ");
for (int i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
}

// Bubble Sort logic
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++)
        { printf("Comparing %i
        and %i\n",arr[j],arr[j+1]); if (arr[j] > arr[j + 1])
        {
            // Swap elements
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
            printf("Swapped %i and %i\n",arr[j],arr[j+1]);
        }
        printf("Current array state: ");
        for(int k=0;k < n;k++){
            printf("%i ",arr[k]);
        }
```

```
            printf("\n");
        }
        printf("After pass %i, array state: ",i+1);
        for(int k = 0;k < n;k++){
            printf("%i ",arr[k]);
        }
        printf("\n");
    }

    // Output the sorted array
    printf("Sorted array in ascending order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

<u>Output :</u>

```
Enter the number of elements in the array: 5
Enter the elements of the array: 2 4 7 3 1
Comparing 2 and 4
Current array state: 2 4 7 3 1
Comparing 4 and 7
Current array state: 2 4 7 3 1
Comparing 7 and 3
Swapped 3 and 7
```

```
Current array state: 2 4 3 7 1
Comparing 7 and 1
Swapped 1 and 7
Current array state: 2 4 3 1 7
After pass 1, array state: 2 4 3 1 7
Comparing 2 and 4
Current array state: 2 4 3 1 7
Comparing 4 and 3
Swapped 3 and 4
Current array state: 2 3 4 1 7
Comparing 4 and 1
Swapped 1 and 4
Current array state: 2 3 1 4 7
After pass 2, array state: 2 3 1 4 7
Comparing 2 and 3
Current array state: 2 3 1 4 7
Comparing 3 and 1
Swapped 1 and 3
Current array state: 2 1 3 4 7
After pass 3, array state: 2 1 3 4 7
Comparing 2 and 1
Swapped 1 and 2
Current array state: 1 2 3 4 7
After pass 4, array state: 1 2 3 4 7
Sorted array in ascending order: 1 2 3 4 7
```

Experiment no : 2

Experiment name : Write a program to find an element using a linear search algorithm.

Title : C program to find an element using a linear search algorithm.

Theory :

❖ The objective of this technique is to search for a specific integer (the "key") within an array of integers provided by the user. This technique employs a linear search algorithm, which checks each element of the array one by one until it finds the desired element or reaches the end of the array.

The problem's logic can be breakdown in 3 steps:

1. Input Phase:

   *The technique prompts the user to enter the number of elements in the array.

   *It then allocates an array of that size and asks for the individual elements.

   *Finally, it prompts the user to enter the element they wish to search for.

2. Searching Phase:

   *The linear search is implemented through a single loop that iterates over each element of the array.

   *For each element, it compares it with the key.

   *If a match is found, it outputs the index and terminates the search.

3. Output Phase:

   *If the key is not found after checking all elements, it informs the user that the element is not present in the array.

Algorithm :

Step 1: Input the Array :

1. Start the program.
2. Prompt the user to enter the number of elements (n) in the array.
3. Read the value of n.
4. Declare an array of size n.
5. Prompt the user to enter n elements of the array.
6. Read and store the elements in the array.

Step 2: Input the Element to Search :

1. Prompt the user to enter the element (key) to search for.
2. Read and store the value of key.
3. Initialize a flag variable (found = 0) to track whether the element is found.

Step 3: Apply Linear Search Algorithm :

1. Loop through the array from index 0 to n-1:

    1. For each element arr[i]**:**
        1. Print the current comparison between arr[i] and key.
        2. If arr[i] is equal to key:
            1. Print that the element is found at index i+1 (1-based index).
            2. Set found = 1.
            3. Exit the loop (break).

Step 4: Output the Search Result :

1. If found remains 0, print that the element is not found in the array.
2. End the program.

Program :

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

```
int main() {
    int n, key, found = 0;

    //Input the size of the array
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];

    //Input the elements of the array
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    //Input the element to search
    printf("Enter the element to search: ");
    scanf("%d", &key);

    //Linear Search Logic
    for (int i = 0; i < n; i++) {
        //Print the current comparison
        printf("Comparing %d with %d\n", arr[i], key);
        if (arr[i] == key) {
            printf("Element %d found at possition %d.\n", key, i+1);
            found = 1;
            break;
        }
    }

    //If the element is not found
    if (!found) {
        print("Element %d not found in the array.\n", key);
    }

    return 0;
}
```

Output :

```
Enter the number of elements in the array: 4
Enter the elements of the array: 5 6 7 9
Enter the element to search: 7
Comparing 5 with 7
Comparing 6 with 7
Comparing 7 with 7
Element 7 found at possition 3.
```

Experiment no : 3

Experiment name : Write a program to sort a linear array using merge sort algorithm.

Title : C program to sort a linear array using merge sort algorithm.

Theory :

❖ Merge Sort is a divide and conquer sorting algorithm that recursively splits an array into smaller subarrays, sorts them, and then merges them back into a sorted sequence. It follows these main steps:

1. Divide

The input array is divided into two halves until each subarray contains a single element.

2. Conquer (Merge Process)

The two sorted halves are merged together in a way that maintains order.

3. Combine

The sorted subarrays are combined recursively until the entire array is sorted.

Let's assume we have an array:

[8, 3, 5, 4, 7, 6, 2, 1]

Step 1: Divide

We split the array recursively:

[8, 3, 5, 4]   [7, 6, 2, 1]

Further breaking down:

[8, 3]   [5, 4]   [7, 6]   [2, 1]

[8] [3]   [5] [4]   [7] [6]   [2] [1]

Each individual element is now considered sorted.

Step 2: Conquer (Merge Process)

We merge back step by step:

[3, 8]   [4, 5]   [6, 7]   [1, 2]

[3, 4, 5, 8]   [1, 2, 6, 7]

[1, 2, 3, 4, 5, 6, 7, 8]

Now the array is completely sorted.

Algorithm :

```
Step 1 : Function: dividee(arr, left, right)
    If left < right:
        Find the middle index mid = (left + right) / 2
        Recursively sort the left half: mergeSort(arr, left, mid)
        Recursively sort the right half: mergeSort(arr, mid+1, right)
        Merge both halves: merge(arr, left, mid, right)
Step 2 : Function: conquer(arr, left, mid, right)
    Create two temporary arrays for left and right halves.
    Compare elements from both halves and insert them into the original
    array in sorted order.
    Copy remaining elements from both halves if any.
Step 3 : Function: main()
    Take input for the array.
    Call mergeSort(arr, 0, n-1), which sorts the array.
    Print the sorted array.
```

Program :

```
#include <stdio.h>
void conquer(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int leftArr[n1], rightArr[n2];

    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

    printf("Left Array: ");
    for (int i = 0; i < n1; i++) printf("%d ", leftArr[i]);
    printf("\n");

    printf("Right Array: ");
    for (int j = 0; j < n2; j++) printf("%d ", rightArr[j]);
    printf("\n");

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        arr[k++] = (leftArr[i] <= rightArr[j]) ? leftArr[i++] : rightArr[j++];
    }
    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];

    printf("Merged Array: ");
    for (int i = left; i <= right; i++) printf("%d ", arr[i]);

    printf("\n");

}


void divide(int arr[], int left, int right) {

    if (left < right) {

        int mid = (left + right) / 2;


        printf("Dividing: ");

        for (int i = left; i <= right; i++) printf("%d ", arr[i]);

        printf("\n");


        divide(arr, left, mid);

        divide(arr, mid + 1, right);

        conquer(arr, left, mid, right);

    }

}


int main() {

    int n;

    printf("Enter the number of elements in the array: ");

    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements of the array: ");

    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    divide(arr, 0, n - 1);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

<u>Output :</u>

```
Enter the number of elements in the array: 6
Enter the elements of the array: 2 4 3 1 0 7
Dividing: 2 4 3 1 0 7
Dividing: 2 4 3
Dividing: 2 4
Left Array: 2
Right Array: 4
Merged Array: 2 4
Left Array: 2 4
Right Array: 3
Merged Array: 2 3 4
Dividing: 1 0 7
```

```
Dividing: 1 0
Left Array: 1
Right Array: 0
Merged Array: 0 1
Left Array: 0 1
Right Array: 7
Merged Array: 0 1 7
Left Array: 2 3 4
Right Array: 0 1 7
Merged Array: 0 1 2 3 4 7
Sorted array: 0 1 2 3 4 7
```

Experiment no : 4

Experiment name : Write a program to find an element using binary search algorithm.

Title : C program to find an element using binary search algorithm.

Theory :

❖ Binary Search is a fundamental algorithm used to find an element in a sorted array efficiently. It follows a divide-and-conquer approach, repeatedly dividing the search range by half until the target element is found or the range becomes empty.

Suppose,We have a sorted array and need to find the position of a given target element using binary search. Instead of checking elements one by one (like linear search), binary search halves the search space in each step, making it significantly faster.

Example:

Given sorted array:

arr=[2,5,8,12,16,23,38,45,57,69]

Target: **16**

Compare 1:

1. left = 0, right = 9
2. mid = (0 + 9) / 2 = 4
3. arr[mid] = arr[4] = 16
4. Match found!

Algorithm :

Step 1 : Start with two pointers:

Left pointer (left = 0) at the beginning.

Right pointer (right = n-1) at the end.

· Calculate the midpoint:

Mid = left + (right - left)/2

Step 2 : Compare arr[mid] with the target:

If arr[mid] == target: Target found at index mid.

If arr[mid] < target: The target must be in the right half, so update left = mid + 1.

If arr[mid] > target: The target must be in the **left half**,

so update right = mid - 1.

Step 3 : Repeat steps **2-3** until left > right:

If the loop ends without finding the target,

it **does not exist** in the array.

.

Program :

```
#include <stdio.h>
int main() {
   int n, target, left, right, mid, step = 1;
   //Taking array size input
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];
   //Taking sorted array input
   printf("Enter %d sorted elements: ", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   //Taking target element input
   printf("Enter the number to search: ");
   scanf("%d", &target);
```

```
    //Binary Search Logic

    left = 0;right = n - 1;

    printf("\nBinary Search Process:\n");

    while (left <= right) {

        mid = left + (right - left) / 2;

        //Print the current step details

        printf("Step %d: Searching in range [%d - %d], Mid = arr[%d]
      = %d\n", step, left+1, right+1, mid+1, arr[mid]);

        printf("Step %d: Comparing target %d with arr[%d] = %d\n", step,
      target, mid+1, arr[mid]);

        if (arr[mid] == target) {

            printf("Target %d found at possition %d\n", target, mid+1);

            return 0;

        } else if (arr[mid] < target) {

            printf("Target is greater than arr[%d], searching right half\n", mid);

            left = mid + 1;

        } else {

            printf("Target is less than arr[%d], searching left half\n", mid);

            right = mid - 1;

        }

        step++;

    }

    printf("Target %d not found in the array\n", target);

    return 0;

}
```

Output :

```
Enter the number of elements: 5

Enter 5 sorted elements: 2 4 6 7 9

Enter the number to search: 9


Binary Search Process:

Step 1: Comparing target 9 with arr[3] = 6

Target is greater than arr[2], searching right half

Step 2: Searching in range [4 - 5], Mid = arr[4] = 7

Step 2: Comparing target 9 with arr[4] = 7

Target is greater than arr[3], searching right half

Step 3: Searching in range [5 - 5], Mid = arr[5] = 9

Step 3: Comparing target 9 with arr[5] = 9

Target 9 found at possition 5
```

Experiment no : 5

Experiment name : Write a program to find a given pattern from text using the pattern matching algorithm.

Title : C program to find a given pattern from text using the pattern matching algorithm.

Theory :

❖ Pattern matching is a fundamental problem in text processing, where we need to find the occurrence(s) of a pattern inside a given text. The naïve pattern matching algorithm checks for the pattern at each possible position in the text and verifies whether it matches character by character.

We are given:

A text (large string).

A pattern (substring) that needs to be searched inside the text.

Naïve Pattern Matching Algorithm Logic:

1. Start at each possible position in the text (i = 0 to textLen - patternLen).
2. Compare characters of the text with the pattern one by one.
3. If all characters match, print the index.
4. If a mismatch occurs, move to the next starting position and repeat.
5. Stop when the pattern is found or the search completes.

Example :

Input : Text: "hello world" & Pattern: "wor"

Checkings :

i = 0 → "hel" ≠ "wor" (Mismatch)

i = 1 → "ell" ≠ "wor" (Mismatch)

i = 2 → "llo" ≠ "wor" (Mismatch)

i = 3 → "lo " ≠ "wor" (Mismatch)

i = 4 → "o w" ≠ "wor" (Mismatch)

i = 5 → " wo" ≠ "wor" (Mismatch)

i = 6 → "wor" == "wor" (Match found at index 6)

Algorithm :

```
Step 1 :Take input for the text and pattern.
Step 2 : Find the length of text and pattern.
Step 3 : Loop through each possible starting index i in text from 0 to
         textLen - patternLen:
             Initialize j = 0.
             Compare characters of pattern[j] with text[i + j]:
                 * If all characters match (j == patternLen),
                   pattern is found at index i. Print the index and stop.
                 * If a mismatch occurs, move to the next
                   starting position and repeat.
Step 4 :  If no match is found after the loop, print "Pattern not found".
```

Program :

```
#include <stdio.h>

#include <string.h>

int main() {
    char text[1000], pattern[1000];

    printf("Enter the text: ");

    fgets(text, sizeof(text), stdin);

    text[strcspn(text, "\n")] = '\0';

    printf("Enter the pattern: ");

    fgets(pattern, sizeof(pattern), stdin);

    pattern[strcspn(pattern, "\n")] = '\0';

    int textLen = strlen(text);

    int patternLen = strlen(pattern);

    // Naive Pattern Matching Algorithm

    for (int i = 0; i <= textLen - patternLen; i++) {

        int j = 0;

        printf("\nChecking substring starting at index %d: ", i);

        while (j < patternLen && text[i + j] == pattern[j]) {

            printf("Comparing text[%d] = '%c' with pattern[%d] = '%c' =>
Match\n", i + j, text[i + j], j, pattern[j]);

            j++;

        }

        if (j == patternLen) {

            printf("Pattern found at index %d\n", i);
            break;
        } else {
            printf("Pattern does not match starting at index %d\n", i);
        }
    }
    return 0;
}
```

Output :

```
Enter the text: hello world
Enter the pattern: wor
Checking substring starting at index 0: Pattern does not match starting at
index 0
Checking substring starting at index 1: Pattern does not match starting at
index 1
Checking substring starting at index 2: Pattern does not match starting at
index 2
Checking substring starting at index 3: Pattern does not match starting at
index 3
Checking substring starting at index 4: Pattern does not match starting at
index 4
```

```
Checking substring starting at index 5: Pattern does not match starting at index 5

Checking substring starting at index 6: Comparing text[6] = 'w' with pattern[0] = 'w' => Match

Comparing text[7] = 'o' with pattern[1] = 'o' => Match

Comparing text[8] = 'r' with pattern[2] = 'r' => Match

Pattern found at index 6
```

Experiment no : 6

Experiment name : Write a program to implement a queue data structure along with it's typical operations.

Title : Python program to implement a queue data structure along with it's typical operations.

Theory :

- A queue is a linear data structure that follows the FIFO (First-In-First-Out) principle. This means that elements are added at the rear (enqueue) and removed from the front (dequeue).

  We can implement such operations on a queue data structure that's are given below :

1. *Enqueue* → Add an element to the rear.
2. *Dequeue* → Remove an element from the front.
3. *Display* → Show all elements in the queue.
4. *Exit* → Stop execution.

Example :

1.initialize: Queue: []

2.Enqueue(10): Add 10 to the queue**.**

Queue: [10]Front: 10, Rear: 10

3.Enqueue(20):Add 20 to the queue.

Queue: [10, 20]Front: 10, Rear: 20

4.Enqueue(30):Add 30 to the queue.

Queue: [10, 20,30]Front: 10, Rear: 30

5.Dequeue():Remove 10 (first element).

Queue: [20,30]Front: 20, Rear: 30

6.Enqueue(40):Add 40 to the queue.

Queue: [20,30,40]Front: 20, Rear: 40

7.Dequeue():Remove 20 .

Queue: [30,40]Front: 30, Rear: 40

8.Display():

Queue elements are: 30 40

Front: 30, Rear: 40

9.Dequeue():Remove 30.

Queue: [40]Front: 40, Rear: 40

10.Dequeue():Remove 40 .

Queue is empty!

11.Dequeue(): (when queue is empty)

Queue is empty! Cannot dequeue.

12.Exit:

Exiting…

Algorithm :

```
Step 1: Initialize an empty queue (queue = []).
Step 2: Loop indefinitely until the user chooses to exit:
    1. Display Menu :
        1. Enqueue2. Dequeue3. Display4. Exit
    1. Take user input (choice).
    2. Perform the corresponding operation:
        1. If choice = 1 (Enqueue):
            1. Take an integer input (value).
            2. Add value to the queue (queue.append(value)).
            3. Print a success message.
        2. If choice = 2 (Dequeue):
            1. If queue is not empty, remove and print the first
               element (queue.pop(0)).
            2. If queue is empty, print "Queue is empty! Cannot
               dequeue."
        3. If choice = 3 (Display):
            1. If queue is not empty, print all elements and show
               front & rear.
            2. If queue is empty, print "Queue is empty!"
        4. If choice = 4 (Exit):
            1. Print "Exiting..." and terminate the program.
        5. If invalid choice, print "Invalid choice! Try again."
    3. Show the current state of the queue after every operation.

Step 3: End of program.
```

Program :

```
def main():
    queue = []  # Initialize an empty queue

    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Display")
        print("4. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:  # Enqueue
            value = int(input("Enter the value to enqueue: "))
            queue.append(value)
            print(f"{value} enqueued to the queue.")

        elif choice == 2:  # Dequeue
            if queue:
                print(f"Dequeued value: {queue.pop(0)}")
            else:
                print("Queue is empty! Cannot dequeue.")

        elif choice == 3:  # Display
            if queue:
                print("Queue elements are:", *queue)
                print(f"Front: {queue[0]}, Rear: {queue[-1]}")
            else:
                print("Queue is empty!")

        elif choice == 4:  # Exit
            print("Exiting...")
            break

        else:
            print("Invalid choice! Try again.")

        # Show current state of the queue after each operation
        if queue:
            print("Current queue:", *queue)
        else:
            print("Queue is empty.")

if __name__ == "__main__":
    main()
```

Output :

```
Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit
```

```
Enter your choice: 1

Enter the value to enqueue: 10

10 enqueued to the queue.

Current queue: 10



Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 20

20 enqueued to the queue.

Current queue: 10 20



Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit
```

```
Enter your choice: 1

Enter the value to enqueue: 30

30 enqueued to the queue.

Current queue: 10 20 30


Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

Dequeued value: 10

Current queue: 20 30


Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3
```

```
Queue elements are: 20 30

Front: 20, Rear: 30

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4

Exiting…
```

Experiment no : 7

Experiment name : Write a program to solve n queen's problem using backtracking.

Title : C program to solve n queen's problem using backtracking.

Theory:

❖ The N-Queens problem is a classic combinatorial problem where the goal is to place N queens on an N×N chessboard such that no two queens threaten each other. This means:

- No two queens can share the same row.

- No two queens can share the same column.
- No two queens can share the same diagonal.

Backtracking is a trial-and-error method used to find solutions by exploring all possible placements of queens on the chessboard. It incrementally builds solutions and backs out when a conflict is detected.

Steps in Backtracking for N-Queens:

1. *Start from Row 0:* Begin by trying to place a queen in the first row.
2. *Try Each Column in a Row:*
    1. For each column in the current row, check if placing a queen would lead to a conflict with any previously placed queens.
    2. If a queen can be safely placed (no conflict in column or diagonal), move to the next row.
3. *Backtrack on Conflict:*

    1. If no valid position is found in a row (due to conflicts), backtrack to the previous row and move the queen to the next column.

4. Repeat this process until all queens are placed on the board without conflict.
5. If a valid solution is found (all queens placed), record the solution.
6. Continue searching for other possible solutions by backtracking and exploring different queen placements.

<u>Algorithm :</u>

Step 1 : Initialize the board:

*Create a 2D array board[N][N] to represent the chessboard.

*Initialize all cells of the board to 0, indicating no queens are placed.

Step 2 : Create a function to check if placing a queen is safe:

*Define a function isSafe(board, row, col):

- Check if there is any queen already placed in the same column (board[row][col]).
- Check if there is any queen in the upper-left diagonal.
- Check if there is any queen in the upper-right diagonal.
- Return true if no conflict is found; otherwise, return false.

Step 3 : Define the backtracking function:

* Define a function placeQueens(board, row):

- If row == N: All queens are placed successfully, print the board and return.
- Otherwise, try placing a queen in every column (col) of the current row:
    - For each col from 0 to N-1:
        - If placing a queen at (row, col) is safe (use isSafe(board, row, col)):
            - Place the queen (board[row][col] = 1).
            - Recursively try to place the next queen by calling placeQueens(board, row + 1).
            - If the next row placement was not successful (backtracking occurs), remove the queen (board[row][col] = 0).

Step 4 : Start the backtracking process:

*Call the placeQueens(board, 0) function to begin placing queens starting from row 0.

Step 5 : Output the solutions:

*Each time all queens are placed correctly (row == N), print the board.

Program :

```
#include <stdio.h>

#define N 4  // Change this for different board sizes

int main() {

   int placed[N], row = 0, col = 0, solutionCount = 0;

   while (row >= 0) {

      int found = 0;

      printf("\nTrying to place a queen in row %d...\n", row);

      while (col < N) {

         printf("Checking (%d, %d)... ", row, col);

         int safe = 1;

         for (int prev = 0; prev < row; prev++) {

            if (placed[prev] == col ||

               placed[prev] - prev == col - row ||

               placed[prev] + prev == col + row) {

               printf("Conflict with queen at (%d, %d)\n", prev, placed[prev]);

               safe = 0;
```

```
                break;
            }
        }
        if (safe) {
            placed[row] = col;
            printf("Placed queen at (%d, %d)\n", row, col);
            row++;
            col = 0;  // Reset for next row
            found = 1;
            break;
        }
        col++;
    }
    if (!found) {
        row--;
        if (row >= 0) {
            col = placed[row] + 1;
            printf("Backtracking to row %d, trying next column %d\n", row,
col);
        }
        }
        if (row == N) {
            printf("\nSolution %d found:\n", ++solutionCount);
            for (int i = 0; i < N; i++, printf("\n"))
                for (int j = 0; j < N; j++)
                    printf("%c ", (placed[i] == j) ? 'Q' : '.');

            row--;  // Backtrack for more solutions
            col = placed[row] + 1;
        }
    }
    printf("\nTotal solutions found: %d\n", solutionCount);
}
```

<u>Output :</u>

```
Trying to place a queen in row 0...
Checking (0, 0)... Placed queen at (0, 0)


Trying to place a queen in row 1...
Checking (1, 0)... Conflict with queen at (0, 0)
Checking (1, 1)... Conflict with queen at (0, 0)

Checking (1, 2)... Placed queen at (1, 2)


Trying to place a queen in row 2...

Checking (2, 0)... Conflict with queen at (0, 0)

Checking (2, 1)... Conflict with queen at (1, 2)

Checking (2, 2)... Conflict with queen at (0, 0)

Checking (2, 3)... Conflict with queen at (1, 2)

Backtracking to row 1, trying next column 3


Trying to place a queen in row 1...

Checking (1, 3)... Placed queen at (1, 3)


Trying to place a queen in row 2...

Checking (2, 0)... Conflict with queen at (0, 0)

Checking (2, 1)... Placed queen at (2, 1)


Trying to place a queen in row 3...

Checking (3, 0)... Conflict with queen at (0, 0)

Checking (3, 1)... Conflict with queen at (1, 3)
Checking (3, 2)... Conflict with queen at (2, 1)
Checking (3, 3)... Conflict with queen at (0, 0)
Backtracking to row 2, trying next column 2

Trying to place a queen in row 2...
Checking (2, 2)... Conflict with queen at (0, 0)
Checking (2, 3)... Conflict with queen at (1, 3)
Backtracking to row 1, trying next column 4

Trying to place a queen in row 1...
Backtracking to row 0, trying next column 1

Trying to place a queen in row 0...
Checking (0, 1)... Placed queen at (0, 1)

Trying to place a queen in row 1...
Checking (1, 0)... Conflict with queen at (0, 1)
Checking (1, 1)... Conflict with queen at (0, 1)
Checking (1, 2)... Conflict with queen at (0, 1)
Checking (1, 3)... Placed queen at (1, 3)
```

```
Trying to place a queen in row 2...

Checking (2, 0)... Placed queen at (2, 0)


Trying to place a queen in row 3...

Checking (3, 0)... Conflict with queen at (2, 0)

Checking (3, 1)... Conflict with queen at (0, 1)

Checking (3, 2)... Placed queen at (3, 2)


Solution 1 found:

. Q . .

. . . Q

Q . . .

. . Q .


Trying to place a queen in row 3...

Checking (3, 3)... Conflict with queen at (1, 3)

Backtracking to row 2, trying next column 1


Trying to place a queen in row 2...
```

```
Checking (2, 1)... Conflict with queen at (0, 1)

Checking (2, 2)... Conflict with queen at (1, 3)

Checking (2, 3)... Conflict with queen at (0, 1)

Backtracking to row 1, trying next column 4


Trying to place a queen in row 1...

Backtracking to row 0, trying next column 2


Trying to place a queen in row 0...

Checking (0, 2)... Placed queen at (0, 2)


Trying to place a queen in row 1...

Checking (1, 0)... Placed queen at (1, 0)


Trying to place a queen in row 2...

Checking (2, 0)... Conflict with queen at (0, 2)

Checking (2, 1)... Conflict with queen at (1, 0)

Checking (2, 2)... Conflict with queen at (0, 2)

Checking (2, 3)... Placed queen at (2, 3)
```

```
Trying to place a queen in row 3...
Checking (3, 0)... Conflict with queen at (1, 0)
Checking (3, 1)... Placed queen at (3, 1)

Solution 2 found:
. . Q .
Q . . .
. . . Q
. Q . .

Trying to place a queen in row 3...
Checking (3, 2)... Conflict with queen at (0, 2)
Checking (3, 3)... Conflict with queen at (2, 3)
Backtracking to row 2, trying next column 4

Trying to place a queen in row 2...
Backtracking to row 1, trying next column 1

Trying to place a queen in row 1...
Checking (1, 1)... Conflict with queen at (0, 2)
Checking (1, 2)... Conflict with queen at (0, 2)

Checking (1, 3)... Conflict with queen at (0, 2)

Backtracking to row 0, trying next column 3


Trying to place a queen in row 0...

Checking (0, 3)... Placed queen at (0, 3)


Trying to place a queen in row 1...

Checking (1, 0)... Placed queen at (1, 0)


Trying to place a queen in row 2...

Checking (2, 0)... Conflict with queen at (1, 0)

Checking (2, 1)... Conflict with queen at (0, 3)

Checking (2, 2)... Placed queen at (2, 2)


Trying to place a queen in row 3...

Checking (3, 0)... Conflict with queen at (0, 3)

Checking (3, 1)... Conflict with queen at (2, 2)

Checking (3, 2)... Conflict with queen at (1, 0)

Checking (3, 3)... Conflict with queen at (0, 3)
```

```
Backtracking to row 2, trying next column 3

Trying to place a queen in row 2...
Checking (2, 3)... Conflict with queen at (0, 3)
Backtracking to row 1, trying next column 1

Trying to place a queen in row 1...
Checking (1, 1)... Placed queen at (1, 1)
Trying to place a queen in row 2...
Checking (2, 0)... Conflict with queen at (1, 1)
Checking (2, 1)... Conflict with queen at (0, 3)
Checking (2, 2)... Conflict with queen at (1, 1)
Checking (2, 3)... Conflict with queen at (0, 3)
Backtracking to row 1, trying next column 2
Trying to place a queen in row 1...
Checking (1, 2)... Conflict with queen at (0, 3)
Checking (1, 3)... Conflict with queen at (0, 3)
Backtracking to row 0, trying next column 4
Trying to place a queen in row 0...
Total solutions found: 2
```

Experiment no : 8

Experiment name : Consider a set S ={ 5,10,12,13,15,18} and d=30.Write a program to solve the sum of subset problem.

Title : C program to solve the sum of subset problem.

Theory :

- ❖ The Subset Sum Problem asks if there exists a subset of a given set of integers whose sum is equal to a specified target sum.

  - *Input*: A set of integers S=S1,S2,…,SN and a target sum T.
  - *Output*: A subset of SSS whose sum equals TTT, if such a subset exists.

Backtracking is a systematic method for solving problems by incrementally building candidates and abandoning those that fail to meet the criteria.

From the Subset Sum Problem, backtracking will explore all subsets by trying to either include or exclude each element of the set, and check if their sum matches the target.

1.Start from an empty subset:

1. Initially, no elements are included in the subset.

2.At each element, decide whether to include it or not:

1. For each element S[i]S[i]S[i], we have two choices:
   1. Include S[i]S[i]S[i] in the current subset.
   2. Exclude S[i]S[i]S[i] from the current subset.

3.Recursive Backtracking:

1. If including an element leads to a sum that exceeds the target, we discard that path (this is where backtracking comes in).
2. If the sum matches the target, we have found a valid subset and can either return or continue to find other solutions.
3. If all elements are considered and no subset sums to the target, we backtrack.

4.Base Case:

1. If we reach the target sum, print or store the current subset.
2. If the sum exceeds the target or we run out of elements to check, return.

5.Backtracking Logic:

1. At each step, we have a binary decision: either include the current element in the subset or skip it.

2. The backtracking algorithm will recursively explore all possible subsets.

3.If at any point, the sum of the current subset exceeds the target,

the algorithm backtracks to the previous state and tries the next

possibility.

Example :

1.Input: S = {3, 1, 2}, d = 4

2.Total Number of Subsets:

2^3 = 8 subsets, generated by masks 0 to 7.

3.Subset Generation:

Mask 0 (000):

Subset: { }

Sum = 0 → Not equal to 4.

Mask 1 (001):

Subset: { 3 }

Sum = 3 → Not equal to 4.

Mask 2 (010):

Subset: { 1 }

Sum = 1 → Not equal to 4.

Mask 3 (011):

Subset: { 1, 3 }

Sum = 4 → Found a valid subset!

Mask 4 (100):

Subset: { 2 }

Sum = 2 → Not equal to 4.

Mask 5 (101):

Subset: { 2, 3 }

Sum = 5 → Not equal to 4.

Mask 6 (110):

Subset: { 1, 2 }

Sum = 3 → Not equal to 4.

Mask 7 (111):

Subset: { 1, 2, 3 }

Sum = 6 → Not equal to 4.

4.Result:

The subset { 1, 3 } has a sum of 4, so it is printed.

Total subsets found = 1.

Algorithm :

Step 1: Input:

1. The user provides the set size N, the elements of the set S[], and the target sum d.

Step 2 : Generate Subsets:

1. We use a bitmask to generate all possible subsets:
    1. 1 << N gives 2^N, the total number of subsets.
    2. We loop from mask = 0 to mask = (2^N - 1).

Step 3 : Subset Generation:

1. For each mask, we check each bit:
    1. If mask & (1 << j) is true, include S[j] in the subset and add it to the sum.

Step 4 : Sum Check:

1. After generating the subset, we check if the sum of the elements in that subset equals the target sum. If so, we print that subset.

Step 5 : Print Result:

1. We print the total count of subsets whose sum is equal to the target sum.

Program :

```
#include <stdio.h>

int main() {

    int N, target_sum;

    // Input for set elements

    printf("Enter the number of elements: ");

    scanf("%d", &N);

    int S[N];

    printf("Enter the elements: ");

    for (int i = 0; i < N; i++) {

        scanf("%d", &S[i]);

    }

    // Input for target sum

    printf("Enter the target sum: ");

    scanf("%d", &target_sum);

    int total_subsets = 1 << N;

    printf("The total subsets of %d is: %d\n", target_sum, total_subsets);

    int count = 0;

    printf("Subsets with sum %d:\n", target_sum);
```

```
for (int mask = 0; mask < total_subsets; mask++) {

    int subset_sum = 0;

    printf("{ ");

    // Generate subset based on current mask

    for (int j = 0; j < N; j++) {

        if (mask & (1 << j)) {

            printf("%d ", S[j]);

            subset_sum += S[j];

        }

    }

    printf("}");

    // Check if the subset sum equals target sum

    if (subset_sum == target_sum) {

        printf(" FOUND");

        count++;

    }

    printf("\n");

}

// Print total count of valid subsets

printf("Total subsets found: %d\n", count);
```

```
    return 0;

}
```

Output :

```
Enter the number of elements: 6

Enter the elements: 5 10 12 13 15 18

Enter the target sum: 30

The total subsets of 30 is: 64

Subsets with sum 30:

{ }

{ 5 }

{ 10 }

{ 5 10 }

{ 12 }

{ 5 12 }

{ 10 12 }

{ 5 10 12 }

{ 13 }

{ 5 13 }

{ 10 13 }
{ 5 10 13 }

{ 12 13 }

{ 5 12 13 } FOUND

{ 10 12 13 }

{ 5 10 12 13 }

{ 15 }

{ 5 15 }

{ 10 15 }

{ 5 10 15 } FOUND

{ 12 15 }

{ 5 12 15 }

{ 10 12 15 }

{ 5 10 12 15 }

{ 13 15 }

{ 5 13 15 }

{ 10 13 15 }

{ 5 10 13 15 }

{ 12 13 15 }

{ 5 12 13 15 }

{ 10 12 13 15 }
{ 5 10 12 13 15 }

{ 18 }

{ 5 18 }

{ 10 18 }

{ 5 10 18 }

{ 12 18 } FOUND

{ 5 12 18 }

{ 10 12 18 }

{ 5 10 12 18 }

{ 13 18 }

{ 5 13 18 }

{ 10 13 18 }

{ 5 10 13 18 }

{ 12 13 18 }

{ 5 12 13 18 }

{ 10 12 13 18 }

{ 5 10 12 13 18 }

{ 15 18 }

{ 5 15 18 }

{ 10 15 18 }
{ 5 10 15 18 }

{ 12 15 18 }

{ 5 12 15 18 }

{ 10 12 15 18 }

{ 5 10 12 15 18 }

{ 13 15 18 }

{ 5 13 15 18 }

{ 10 13 15 18 }

{ 5 10 13 15 18 }

{ 12 13 15 18 }

{ 5 12 13 15 18 }

{ 10 12 13 15 18 }

{ 5 10 12 13 15 18 }

Total subsets found: 3
```

Experiment no : 9

Experiment name : Write a program to solve the following 0/1 knapsack using dynamic programming approach profits P = (15,25,13,23) , weight w=(2,6,12,9), Knapsack C=20 and number of item n=4.

Title : Python program to solve the following 0/1 knapsack using dynamic programming approach.

Theory:

❖ The 0/1 Knapsack problem is a combinatorial optimization problem where we have n items, each with a given profit and weight, and a knapsack with a fixed capacity. The goal is to maximize the total profit without exceeding the knapsack's capacity. Each item can either be included (1) or excluded (0), hence the name 0/1 Knapsack.

To solve the problem we can build some logic which are given below :

1: Understand the Problem

- n items, each with a profit (pi) and weight (wi) .
- A knapsack with capacity W.
- We must maximize the total profit while ensuring that the total weight does not exceed W.
- Each item can be either included (1) or not included (0).

2: Define the State

Let dp[i][w] represent the maximum profit achievable using the first i items and a knapsack capacity of w.

3: Define the Recurrence Relation

For each item i, we have two choices:

I)Exclude the item → Profit remains the same as without this item:

dp[i][w]=dp[i−1][w]

II)Include the item (if it fits in the remaining capacity) → Add the profit of this item and solve for reduced capacity:

$$dp[i][w] = pi + dp[i-1][w-wi]$$

Thus, we take the maximum of both choices:

$$dp[i][w] = \max(dp[i-1][w], (pi + dp[i-1][w-wi]))$$

Condition: We include the item only if its weight does not exceed the capacity ($wi \leq w$).

4: Base Case Initialization

- If no items are available ($i=0$), the maximum profit is 0 for all capacities. $dp[0][w]=0 \quad \forall w$
- If capacity is 0 ($w=0$), the maximum profit is 0 for all items. $dp[i][0]=0 \ \forall I$

5: Fill the DP Table (Bottom-Up Approach)

1. Create a 2D table dp[n+1][W+1].(where rows represent items, and columns represent capacities).
2. Iterate over items (1 to nnn).
3. Iterate over capacity (1 to W).
4. Use the recurrence relation to compute the maximum profit.

6: Find the Selected Items (Backtracking the Solution)

1. Start from dp[n][W], which contains the maximum profit.
2. If $dp[i][w] \neq dp[i-1][w]$, item i was included.
3. Subtract the item's weight from w and repeat until w=0.

7: Return the Result

The final answer (maximum profit) is stored in:

dp[n][W]

Example :

Input :

item	profit(pi)	weight(wi)
1	60	2
2	100	3
3	120	5

Knapsack Capacity: W = 5

DP Table :

Capacity =>	0	1	2	3	4	5
Item 0	0	0	0	0	0	0
Item 1	0	0	60	60	60	60
Item 2	0	0	60	100	100	160
Item 3	0	0	60	100	100	160

Final Maximum Profit: 160.

Selected Items: Item 1 (60) and Item 2 (100).

<u>Algorithm :</u>

Step 1: Take Input

1. Read the number of items (n).
2. Initialize two lists:
    1. profits[] → Stores the profit of each item.
    2. weights[] → Stores the weight of each item.
3. Take input for the profit of each item and store it in profits[].
4. Take input for the weight of each item and store it in weights[].

5. Read the capacity WWW of the knapsack.

Step 2: Initialize the DP Table

1. Create a 2D array dp[][] of size (n+1) × (W+1) initialized to 0.
    1. dp[i][w] represents the maximum profit achievable using the first i items and knapsack capacity w.

Step 3: Fill the DP Table (Bottom-Up Approach)

1. Iterate through items (i = 1 to n):
    1. Iterate through capacities (w = 1 to W):
        1. If the item can fit (wi ≤ w):
            1. Compute two options:
                1. Include the item → Profit = profits[i-1] + dp[i-1][w - weights[i-1]]
                2. Exclude the item → Profit = dp[i-1][w]
            2. Take the maximum of both and store in dp[i][w].
        2. Else (item too heavy to fit):
            1. Carry forward the previous value: dp[i][w] = dp[i-1][w].
        3. Print decision-making process (which item is considered and included/excluded).

Step 4: Display the DP Table

1. Print the entire dp table for clarity.

Step 5: Find the Selected Items (Backtracking)

1. Start from the last cell dp[n][W] (this contains the maximum profit).
2. Iterate backward from n to 1:
    1. If dp[i][w] != dp[i-1][w], the item was included.
    2. Print the item's profit and weight.
    3. Reduce the remaining capacity w -= weights[i-1].
3. Stop when w = 0.

Step 6: Print the Maximum Profit

1. Print dp[n][W], which gives the maximum profit that can be obtained.

<u>Program :</u>

```
def knapsack():

    n = int(input("Enter the number of items: "))

    profits, weights = [], []

    # Input profits and weights

    for i in range(n):

        profits.append(int(input(f"Profit of item {i + 1}: ")))

    for j in range(n):

        weights.append(int(input(f"Weight of item {j + 1}: ")))

    capacity = int(input("Enter the capacity of the knapsack: "))
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]


    # Dynamic Programming table filling
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                include = profits[i - 1] + dp[i - 1][w - weights[i - 1]]
                exclude = dp[i - 1][w]
                dp[i][w] = max(include, exclude)
                # Print decision-making process
                print(f"Considering item {i} (Profit: {profits[i - 1]}, Weight: {weights[i - 1]}) with capacity {w}")
                print(f"Include item {i}: {profits[i - 1]} + dp[{i - 1}][{w - weights[i - 1]}] = {dp[i][w]}")
                print(f"Exclude item {i}: dp[{i - 1}][{w}] = {exclude}")
            else:
                dp[i][w] = dp[i - 1][w]
                print(f"Item {i} too heavy for capacity {w}, dp[{i}][{w}] = {dp[i][w]}")
            print(f"dp[{i}][{w}] = {dp[i][w]}\n")
    # Display DP table
    print("DP Table:")
    for row in dp:
        print(" ".join(f"{x:2d}" for x in row))
    # Backtrack to find selected items
    w = capacity
    print("\nItems included for maximum profit:")
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Item is included
            print(f"Item {i} (Profit: {profits[i - 1]}, Weight: {weights[i - 1]})")
            w -= weights[i - 1]
    print(f"Maximum profit in the knapsack: {dp[n][capacity]}")
if __name__ == "__main__":
    knapsack()
```

<u>Output :</u>

```
Enter the number of items: 4
Profit of item 1: 15
Profit of item 2: 25
Profit of item 3: 13
Profit of item 4: 23

Weight of item 1: 2

Weight of item 2: 6

Weight of item 3: 12

Weight of item 4: 9

Enter the capacity of the knapsack: 20

Item 1 too heavy for capacity 1, dp[1][1] = 0

dp[1][1] = 0



Considering item 1 (Profit: 15, Weight: 2) with capacity 2

Include item 1: 15 + dp[0][0] = 15

Exclude item 1: dp[0][2] = 0

dp[1][2] = 15



Considering item 1 (Profit: 15, Weight: 2) with capacity 3

Include item 1: 15 + dp[0][1] = 15

Exclude item 1: dp[0][3] = 0

dp[1][3] = 15



Considering item 1 (Profit: 15, Weight: 2) with capacity 4
```

Include item 1: 15 + dp[0][2] = 15

Exclude item 1: dp[0][4] = 0

dp[1][4] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 5

Include item 1: 15 + dp[0][3] = 15

Exclude item 1: dp[0][5] = 0

dp[1][5] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 6

Include item 1: 15 + dp[0][4] = 15

Exclude item 1: dp[0][6] = 0

dp[1][6] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 7

Include item 1: 15 + dp[0][5] = 15

Exclude item 1: dp[0][7] = 0

dp[1][7] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 8

Include item 1: 15 + dp[0][6] = 15

Exclude item 1: dp[0][8] = 0

dp[1][8] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 9

Include item 1: 15 + dp[0][7] = 15

Exclude item 1: dp[0][9] = 0

dp[1][9] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 10

Include item 1: 15 + dp[0][8] = 15

Exclude item 1: dp[0][10] = 0

dp[1][10] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 11

Include item 1: 15 + dp[0][9] = 15

Exclude item 1: dp[0][11] = 0

dp[1][11] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 12

Include item 1: 15 + dp[0][10] = 15

Exclude item 1: dp[0][12] = 0

dp[1][12] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 13

Include item 1: 15 + dp[0][11] = 15

Exclude item 1: dp[0][13] = 0

dp[1][13] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 14

Include item 1: 15 + dp[0][12] = 15

Exclude item 1: dp[0][14] = 0

dp[1][14] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 15

Include item 1: 15 + dp[0][13] = 15

Exclude item 1: dp[0][15] = 0

dp[1][15] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 16

Include item 1: 15 + dp[0][14] = 15

Exclude item 1: dp[0][16] = 0

dp[1][16] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 17

Include item 1: 15 + dp[0][15] = 15

Exclude item 1: dp[0][17] = 0

dp[1][17] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 18

Include item 1: 15 + dp[0][16] = 15

Exclude item 1: dp[0][18] = 0

dp[1][18] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 19

Include item 1: 15 + dp[0][17] = 15

Exclude item 1: dp[0][19] = 0

dp[1][19] = 15

Considering item 1 (Profit: 15, Weight: 2) with capacity 20

```
Include item 1: 15 + dp[0][18] = 15

Exclude item 1: dp[0][20] = 0

dp[1][20] = 15


Item 2 too heavy for capacity 1, dp[2][1] = 0

dp[2][1] = 0


Item 2 too heavy for capacity 2, dp[2][2] = 15

dp[2][2] = 15


Item 2 too heavy for capacity 3, dp[2][3] = 15

dp[2][3] = 15


Item 2 too heavy for capacity 4, dp[2][4] = 15

dp[2][4] = 15


Item 2 too heavy for capacity 5, dp[2][5] = 15

dp[2][5] = 15


Considering item 2 (Profit: 25, Weight: 6) with capacity 6
```

Include item 2: 25 + dp[1][0] = 25

Exclude item 2: dp[1][6] = 15

dp[2][6] = 25

Considering item 2 (Profit: 25, Weight: 6) with capacity 7

Include item 2: 25 + dp[1][1] = 25

Exclude item 2: dp[1][7] = 15

dp[2][7] = 25

Considering item 2 (Profit: 25, Weight: 6) with capacity 8

Include item 2: 25 + dp[1][2] = 40

Exclude item 2: dp[1][8] = 15

dp[2][8] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 9

Include item 2: 25 + dp[1][3] = 40

Exclude item 2: dp[1][9] = 15

dp[2][9] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 10

Include item 2: 25 + dp[1][4] = 40

Exclude item 2: dp[1][10] = 15

dp[2][10] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 11

Include item 2: 25 + dp[1][5] = 40

Exclude item 2: dp[1][11] = 15

dp[2][11] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 12

Include item 2: 25 + dp[1][6] = 40

Exclude item 2: dp[1][12] = 15

dp[2][12] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 13

Include item 2: 25 + dp[1][7] = 40

Exclude item 2: dp[1][13] = 15

dp[2][13] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 14

Include item 2: 25 + dp[1][8] = 40

Exclude item 2: dp[1][14] = 15

dp[2][14] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 15

Include item 2: 25 + dp[1][9] = 40

Exclude item 2: dp[1][15] = 15

dp[2][15] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 16

Include item 2: 25 + dp[1][10] = 40

Exclude item 2: dp[1][16] = 15

dp[2][16] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 17

Include item 2: 25 + dp[1][11] = 40

Exclude item 2: dp[1][17] = 15

dp[2][17] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 18

Include item 2: 25 + dp[1][12] = 40

Exclude item 2: dp[1][18] = 15

dp[2][18] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 19

Include item 2: 25 + dp[1][13] = 40

Exclude item 2: dp[1][19] = 15

dp[2][19] = 40

Considering item 2 (Profit: 25, Weight: 6) with capacity 20

Include item 2: 25 + dp[1][14] = 40

Exclude item 2: dp[1][20] = 15

dp[2][20] = 40

Item 3 too heavy for capacity 1, dp[3][1] = 0

dp[3][1] = 0

Item 3 too heavy for capacity 2, dp[3][2] = 15

dp[3][2] = 15

Item 3 too heavy for capacity 3, dp[3][3] = 15

dp[3][3] = 15

Item 3 too heavy for capacity 4, dp[3][4] = 15

dp[3][4] = 15

Item 3 too heavy for capacity 5, dp[3][5] = 15

dp[3][5] = 15

Item 3 too heavy for capacity 6, dp[3][6] = 25

dp[3][6] = 25

Item 3 too heavy for capacity 7, dp[3][7] = 25

dp[3][7] = 25

Item 3 too heavy for capacity 8, dp[3][8] = 40

dp[3][8] = 40

Item 3 too heavy for capacity 9, dp[3][9] = 40

dp[3][9] = 40

```
Item 3 too heavy for capacity 10, dp[3][10] = 40
dp[3][10] = 40

Item 3 too heavy for capacity 11, dp[3][11] = 40
dp[3][11] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 12
Include item 3: 13 + dp[2][0] = 40
Exclude item 3: dp[2][12] = 40
dp[3][12] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 13
Include item 3: 13 + dp[2][1] = 40
Exclude item 3: dp[2][13] = 40
dp[3][13] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 14
Include item 3: 13 + dp[2][2] = 40
Exclude item 3: dp[2][14] = 40
```

dp[3][14] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 15

Include item 3: 13 + dp[2][3] = 40

Exclude item 3: dp[2][15] = 40

dp[3][15] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 16

Include item 3: 13 + dp[2][4] = 40

Exclude item 3: dp[2][16] = 40

dp[3][16] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 17

Include item 3: 13 + dp[2][5] = 40

Exclude item 3: dp[2][17] = 40

dp[3][17] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 18

Include item 3: 13 + dp[2][6] = 40

Exclude item 3: dp[2][18] = 40

dp[3][18] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 19

Include item 3: 13 + dp[2][7] = 40

Exclude item 3: dp[2][19] = 40

dp[3][19] = 40

Considering item 3 (Profit: 13, Weight: 12) with capacity 20

Include item 3: 13 + dp[2][8] = 53

Exclude item 3: dp[2][20] = 40

dp[3][20] = 53

Item 4 too heavy for capacity 1, dp[4][1] = 0

dp[4][1] = 0

Item 4 too heavy for capacity 2, dp[4][2] = 15

dp[4][2] = 15

Item 4 too heavy for capacity 3, dp[4][3] = 15

dp[4][3] = 15

Item 4 too heavy for capacity 4, dp[4][4] = 15

dp[4][4] = 15

Item 4 too heavy for capacity 5, dp[4][5] = 15

dp[4][5] = 15

Item 4 too heavy for capacity 6, dp[4][6] = 25

dp[4][6] = 25

Item 4 too heavy for capacity 7, dp[4][7] = 25

dp[4][7] = 25

Item 4 too heavy for capacity 8, dp[4][8] = 40

dp[4][8] = 40

Considering item 4 (Profit: 23, Weight: 9) with capacity 9

Include item 4: 23 + dp[3][0] = 40

Exclude item 4: dp[3][9] = 40

dp[4][9] = 40

```
Considering item 4 (Profit: 23, Weight: 9) with capacity 10

Include item 4: 23 + dp[3][1] = 40

Exclude item 4: dp[3][10] = 40

dp[4][10] = 40


Considering item 4 (Profit: 23, Weight: 9) with capacity 11

Include item 4: 23 + dp[3][2] = 40

Exclude item 4: dp[3][11] = 40

dp[4][11] = 40


Considering item 4 (Profit: 23, Weight: 9) with capacity 12

Include item 4: 23 + dp[3][3] = 40

Exclude item 4: dp[3][12] = 40

dp[4][12] = 40


Considering item 4 (Profit: 23, Weight: 9) with capacity 13

Include item 4: 23 + dp[3][4] = 40

Exclude item 4: dp[3][13] = 40

dp[4][13] = 40
```

Considering item 4 (Profit: 23, Weight: 9) with capacity 14

Include item 4: 23 + dp[3][5] = 40

Exclude item 4: dp[3][14] = 40

dp[4][14] = 40

Considering item 4 (Profit: 23, Weight: 9) with capacity 15

Include item 4: 23 + dp[3][6] = 48

Exclude item 4: dp[3][15] = 40

dp[4][15] = 48

Considering item 4 (Profit: 23, Weight: 9) with capacity 16

Include item 4: 23 + dp[3][7] = 48

Exclude item 4: dp[3][16] = 40

dp[4][16] = 48

Considering item 4 (Profit: 23, Weight: 9) with capacity 17

Include item 4: 23 + dp[3][8] = 63

Exclude item 4: dp[3][17] = 40

dp[4][17] = 63

```
Considering item 4 (Profit: 23, Weight: 9) with capacity 18

Include item 4: 23 + dp[3][9] = 63

Exclude item 4: dp[3][18] = 40

dp[4][18] = 63


Considering item 4 (Profit: 23, Weight: 9) with capacity 19

Include item 4: 23 + dp[3][10] = 63

Exclude item 4: dp[3][19] = 40

dp[4][19] = 63


Considering item 4 (Profit: 23, Weight: 9) with capacity 20

Include item 4: 23 + dp[3][11] = 63

Exclude item 4: dp[3][20] = 53

dp[4][20] = 63


DP Table:

 0 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0

 0  0 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15

 0  0 15 15 15 15 25 25 40 40 40 40 40 40 40 40 40 40 40 40 40
 0  0 15 15 15 15 25 25 40 40 40 40 40 40 40 40 40 40 40 40 53

 0  0 15 15 15 15 25 25 40 40 40 40 40 40 40 48 48 63 63 63 63


Items included for maximum profit:

Item 4 (Profit: 23, Weight: 9)

Item 2 (Profit: 25, Weight: 6)

Item 1 (Profit: 15, Weight: 2)

Maximum profit in the knapsack: 63
```

Experiment no : 10

Experiment name : Write a program to solve the Tower of Hanoi problem for N disk.

Title : Python program to solve the Tower of Hanoi problem for N disk.

Theory :

- ❖ The Tower of Hanoi is a classic recursive problem that involves moving n disks from a source peg to a destination peg using an auxiliary peg, following these rules:

1. Only one disk can be moved at a time.

2. A larger disk cannot be placed on a smaller disk.
3. A disk must always be placed on one of the three pegs.

The minimum number of moves required to solve the problem is:

T(n) = 2^n - 1 [where n is the number of disks.]

Base Case (n = 1):

If there is only one disk, move it directly from the source peg to the destination peg.

Recursive Case (n > 1):

To move n disks from peg A (source) to peg C (destination) using peg B (auxiliary):

1. Move n−1disks from A to B (using C as auxiliary).
2. Move the nth disk from A to C.
3. Move the n−1 disks from B to C (using A as auxiliary).

Example :

1.Suppose Initially:

- Disks: 3
- Pegs: A (Source), C (Destination), B (Auxiliary)

2.Recursive Calls Breakdown

1. Move 2 disks from A → B (using C).
    - Move 1st disk from A → C.
    - Move 2nd disk from A → B.
    - Move 1st disk from C → B.

2. Move 3rd disk from A → C.
3. Move 2 disks from B → C (using A).

    - Move 1st disk from B → A.
    - Move 2nd disk from B → C.
    - Move 1st disk from A → C.

3.Final Moves for n = 3

1. Move disk 1 from A → C
2. Move disk 2 from A → B
3. Move disk 1 from C → B
4. Move disk 3 from A → C
5. Move disk 1 from B → A
6. Move disk 2 from B → C
7. Move disk 1 from A → C

Total Moves: 2^3 - 1 = 7

Algorithm :

Step 1: Read Input

1. Read the number of disks n from the user.
2. Initialize the three pegs:
    1. Source peg: A
    2. Destination peg: C
    3. Auxiliary peg: B

Step 2: Calculate Total Moves

1. Compute the total number of moves using the formula:

T(n) = 2^n - 1

2. Print the total number of moves.

Step 3: Solve Using Recursion

1. Define a recursive function towers(n,source,auxiliary,target):
    1. Base Case: If n=1:
        1. Move disk 1 from source to target.
        2. Return.
    2. Recursive Case: If n>1:
        1. Move n−1 disks from source to auxiliary using target as auxiliary.
        2. Move the nth disk from source to target.
        3. Move n−1 disks from auxiliary to target using source as auxiliary.

Step 4: Execute Recursive Function

1. Call towers(n, 'A', 'C', 'B') to start the process.

Step 5: Display Moves

Print each move in the format:

Move disk X from peg Y to peg Z

where:

1. X = Disk number
2. Y = Source peg
3. Z = Destination peg

Step 6: End Program

1. Stop when all disks are moved to the destination peg.

Program :

```
def hanoi(n, source, target, auxiliary):

    if n == 1:

        print(f"Move disk 1 from {source} to {target}")

        return

    hanoi(n - 1, source, auxiliary, target)

    print(f"Move disk {n} from {source} to {target}")

    hanoi(n - 1, auxiliary, target, source)

n = int(input("Enter number of disks: "))

print(f"Total moves required: {2**n - 1}")

print(f"The sequence of moves involved in the Tower of Hanoi are:")

hanoi(n, 'A', 'C', 'B')  # Pegs : A (source), C (target), B (auxiliary)
```

Output :

```
Enter number of disks: 3

Total moves required: 7

The sequence of moves involved in the Tower of Hanoi are:
```

```
Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C
```

THE END