# 

|  |  |
| --- | --- |
| Index | Title |
| 01 | Write a program to sort a linear array using the bubble sort algorithm |
| 02 | Write a program to find an element using a linear search algorithm |
| 03 | Write a program to sort a linear array using the merge sort algorithm |
| 04 | Write a program to find an element using the binary search algorithm. |
| 05 | Write a program to find a given pattern form text using the pattern matching algorithm. |
| 06 | Write a program to implement a queue data structure along with its typical operations. |
| 07 | Write a program to solve n queen’s problem using backtracking |
| 08 | Consider a set S = ( 5,10,12,13,15,18 ) and d = 30. Write a program to solve the sum of subset problem. |
| 09 | 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 the number of items n = 4. |
| 10 | Write a program to solve the Tower of Hanoi problem for the N disk. |

# Problem number : 01

## Title : Write a program to sort a linear array using the bubble sort algorithm.

# Illustration of the problem :

A list is repeatedly stepped through by the Bubble Sort algorithm, which compares neighboring elements and swaps them if they are out of order. Until the list is sorted, this process is repeated.

This is an explanation of the Bubble Sort reasoning:  
  
Repeatedly loop through the array: Compare neighboring elements for each iteration of the list.

Examine neighboring elements: Compare each entry in the list with its adjacent element.

If necessary, swap: To put them in the right order, swap them if the current element is higher than the subsequent one.

Reduce comparisons to maximize: The largest element "bubbles up" to its proper position after each pass. Therefore, we can disregard the final sorted elements in the subsequent pass, lowering the number of comparisons.

Repeat until sorted: Keep going through the list until there are no swaps in a full pass, indicating that the list is sorted.

#Time complexity: O(n^2)

#Space complexity: O(1)

# Algorithm :

BubbleSort(data,n)

here data is an array with n elements. This algorithm sorts the elements in data.

Step-1(Initialize) :

Define a vector (or array) with n entries that needs to be sorted, v.

Determine v's size as n.

Step-2(Outer Loop (for loop on i from 0 to n-2)) :

Iterate up to the final unsorted entry for each iteration of the list.

The current pass count is represented by the variable i, and the number of unsorted elements decreases by i for every pass that sorts one element to its proper location.

Step-3(Inner Loop (for loop on j from 0 to n-i-2)) :

Examine neighboring elements in the array's unsorted section.

Verify whether v[j] > v[j + 1] for every element v[j] and the subsequent element v[j + 1]:  
To put v[j] and v[j + 1] in ascending order, swap them if true.

Until the end of the unsorted part, keep doing this comparison and switching throughout the inner loop.

Step-4(Repeat steps 2 and 3) :

Reduce the unsorted range by increasing i since the largest unsorted element is "bubbled up" to its proper location after each pass.

Until every piece is in the proper sequence, keep looping.

Step-5(Results) :

After sorting, iterate through each element in v and output the sorted array.

# Source code :

#include <bits/stdc++.h>

using namespace std;

void printVector(const vector<int> &v) {

    for (int num : v) {

        cout << num << " ";

    }

    cout << endl;

}

void bubbleSort(vector<int> &v, int n) {

    for (int i = 0; i < n - 1; i++) {

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

            if (v[j] > v[j + 1]) {

                swap(v[j], v[j + 1]);

                printVector(v); // Print after each swap

            }

        }

    }

}

int main() {

    int n;

    cin >> n;

    vector<int> v(n);

    for (int i = 0; i < n; i++) {

        cin >> v[i];

    }

    cout << "Initial Array: ";

    printVector(v); // Print initial array

    bubbleSort(v, n);

    cout << "Sorted Array: ";

    printVector(v); // Print final sorted array

    return 0;

}

# Sample Input 0:

5

4 5 1 2 3

# Sample Output 0:

1 2 3 4 5

# Sample Input 1:

8

8 4 19 2 7 13 5 16

# Sample Output 1:

2 4 5 7 8 13 16 19

# Problem number : 02

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

# Illustration of the problem :

Suppose we wish to locate a particular item from a list of things that are arranged in a row. Anything from names to numbers to other kinds of data could be represented by this list. Let's say it's a list of numbers for simplicity's sake.

Sequential Checking : Examine each item in order, starting at the beginning.  
Stop When Found : Return the target's location if it is located.  
If Not Found : Return that the target is not in the list if you get to the end without finding it.  
Efficiency : Because linear search examines each element individually, it can be slow for large lists yet effective for tiny lists or unsorted data.

# Algorithm :

Until the target element is located or the full list has been searched, a linear search method sequentially checks each element in a list or array. Here's a detailed breakdown of how to use linear search to locate an element:

Step-1 (Set the index's initial values) :

Begin with the initial element of the list at index0.

Step-2 (Go over each component one by one) :

For every item in the list at index i :

Compare the target and the element arr[i].

Step-3 (Look for a match) :

If arr[i] equals target, the target element is identified by the position (i+1) and declare that Found.

Step-4 (Proceed if there is no correspondence) :

Proceed to the following element (increase the index i by 1) if arr[i] does not match the target.

Step-5 (The list's conclusion) :

Return “Not Found” if the target cannot be located after traversing the full list.

## Complexity of Time:

Best Case: The key may appear at the first index in the best case scenario. Therefore, O(1) is the best case complexity.   
  
Worst situation: the key may be located at the last index, which is the opposite of the end of the list from whence the search began. Therefore, O(N), where N is the list's size, is the worst-case complexity.   
  
Average Situation: O(N)

## Auxiliary Space:

O(1) since only the variable for iterating through the list is utilized, with no additional variables needed.

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <bits/stdc++.h>

using namespace std;

int main()

{

     int n;

     cin >> n;

     vector<int> arr(n);

     for (int i = 0; i < n; i++)

     {

          cin >> arr[i];

     }

     int target;

     cin >> target;

     bool flag = 0;

     for (int i = 0; i < n; i++)

     {

          if (arr[i] == target)

          {

               cout << "Posision: " << i + 1 << endl;

               flag = 1;

          }

     }

     if (flag == 0)

          cout << "Not Found" << endl;

     return 0;

}

## Sample Input 0:

10

31 62 87 70 90 50 55 35 85 72

55

## Sample Output 0:

Found and Position is 7

## Sample Input 1:

10

31 62 87 70 90 50 55 35 85 72

52

## Sample Output 1:

Not Found

# Problem number : 03

## Title : Write a program to sort a linear array using the merge sort algorithm.

# Illustration of the problem :

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by recursively dividing it into two halves, sorting each half, and then merging the two sorted halves into a single sorted array. Here’s a theoretical breakdown:

## 1. Divide:

If the array has more than one element, split the array into two halves.

This division continues recursively until each subarray contains only a single element. A single-element array is trivially sorted.

## 2. Conquer:

Once the array is divided into subarrays of size 1, the algorithm begins to merge them back together.

During the merge step, two sorted subarrays are combined into a single sorted array by comparing the elements of the subarrays one by one and placing the smaller (or larger, depending on sorting order) element in the resulting array.

## 3. Combine:

Once all subarrays have been merged back into a fully sorted array, the process ends.

Merge Sort Algorithm:

## Base Case:

If the array has one element or is empty, return it (as it is already sorted).

Recursive Step:

Divide the array into two halves.

Recursively sort both halves.

Merge the two sorted halves.

Time Complexity:

* Best, Average, Worst-case: O(n log n)

> The array is divided in half recursively, taking **log N** steps.

> Each merge step takes **O(N)** time to combine two halves.

Space Complexity:

* O(n)

# Algorithm :

Step-1: If the array has one or zero elements, return it (base case).

Step-2: Divide the array into two halves:

Find the middle index.

Split the array into left\_half and right\_half.

Step-3: Recursively apply merge\_sort to both halves.

Step-4: Merge the sorted halves:

Compare elements of left\_half and right\_half, placing the smaller one into the original array.

If any elements remain, copy them into the array.

Step-5: Return the sorted array.

# Source code :

## # this code is written by APURBO SHARMA (220604)

#include <bits/stdc++.h>

using namespace std;

// Function to print the current state of the array

void printVector(int ar[], int n) {

    for (int i = 0; i < n; i++)

        cout << ar[i] << " ";

    cout << endl;

}

// Function to merge two sorted halves of the array

void merge(int ar[], int l, int m, int r, int n) {

    int ls = m - l + 1;  // Size of left subarray

    int rs = r - m;      // Size of right subarray

    // Create temporary arrays to hold the left and right subarrays

    vector<int> L(ls), R(rs);

    // Copy data to temporary arrays L[] and R[]

    for (int i = 0; i < ls; i++)

        L[i] = ar[l + i];

    for (int i = 0; i < rs; i++)

        R[i] = ar[m + 1 + i];

    int cur = l, i = 0, j = 0;

    // Merge the two subarrays back into the main array

    while (i < ls && j < rs) {

        if (L[i] <= R[j])

            ar[cur++] = L[i++];

        else

            ar[cur++] = R[j++];

    }

    // Copy any remaining elements of L[]

    while (i < ls)

        ar[cur++] = L[i++];

    // Copy any remaining elements of R[]

    while (j < rs)

        ar[cur++] = R[j++];

    // Print the array after merging two subarrays

    printVector(ar, n);

}

// Function to perform Merge Sort recursively

void mergeSort(int ar[], int l, int r, int n) {

    if (l >= r) // Base case: if left index is greater or equal to right, stop recursion

        return;

    int m = l + (r - l) / 2; // Calculate the middle index

    // Recursively sort the first half

    mergeSort(ar, l, m, n);

    // Recursively sort the second half

    mergeSort(ar, m + 1, r, n);

    // Merge the two sorted halves

    merge(ar, l, m, r, n);

}

int main() {

    int n;

    cin >> n; // Input size of array

    vector<int> ar(n);

    // Input array elements

    for (int i = 0; i < n; i++)

        cin >> ar[i];

    // Print the initial array before sorting

    cout << "Initial Array: ";

    printVector(ar.data(), n);

    // Call mergeSort function

    mergeSort(ar.data(), 0, n - 1, n);

    // Print the sorted array

    cout << "Sorted Array: ";

    printVector(ar.data(), n);

    return 0;

}

## Sample Input :

Enter the elements of the array separated by space: 3 2 1 31 1

## Sample output :

Sorted Array: [1, 1, 2, 3, 31]

# Problem number : 04

## Title : Write a program to find an element using the binary search algorithm.

# Illustration of the problem :

Binary search is an algorithm designed to locate the index of a specific value in a sorted array. It operates by continuously splitting the search range in half until the target value is identified or the range is depleted. The search range is reduced by comparing the target element to the middle value of the current search area.

Requirements for implementing Binary Search Algorithm in a Data Structure

To utilize the Binary Search algorithm:

The data structure needs to be in a sorted order.

Accessing any element within the data structure should occur in constant time.

# Algorithm :

Here is a detailed explanation of this program's algorithm:

That function I use for searching with three parameters binarySearch(arr, size, target)

## Step-1(Enter the Size of the Array ,n) :

Enter n, an integer that denotes the array's element count.

## Step-2(Declare the array and initialize it) :

Set up a vector arr of size n to store the array's contents.

## Step-3 (Step-2(Declare the array and initialize it) :

Each element entered by the user is stored in the arr vector using a loop.

## Step-4 (Add the Target Element here) :

Assume that the value we wish to look for in the array is an integer input target.

## Step-5 (To sort the array) :

To sort the array in ascending order, use sort(arr.begin(), arr.end()). Sorting is required if the array isn't already sorted because binary search requires a sorted array.

## Step-6 (Execute a Binary Search) :

Use the arguments arr, n, and target to invoke the binarySearch function. This function will:

Set the search boundaries by initializing the left to 0 and the right to size -1.

Start a loop that keeps going as long as left <= right, which indicates that there are still items in the search range.

To prevent overflow, compute the midpoint as left + (right - left) / 2.

Verify that arr[mid] equals the target:  
Return mid (target index) if it is.

Move left to mid + 1 (limit the search range to the right half) if arr[mid] is less than the target.

Move right to mid - 1 (limit the search range to the left half) if arr[mid] is bigger than target.

Return -1 to indicate that the target is not in the array if the loop terminates without finding it.

## Step-7 (Output the outcome) :

Verify the binarySearch result:

Output the index where the element was located if the result is not -1.

Indicate that the element was not located in the array if the result is -1.

By breaking down the algorithm into logical processes, this methodical approach makes it possible to perform binary search on a sorted array and efficiently locate the target element (if it exists) in O(logn) time complexity.

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <bits/stdc++.h>

using namespace std;

int binarySearch(vector<int> arr, int size, int target)

{

     int left = 0;

     int right = size - 1;

     while (left <= right)

     {

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

          if (arr[mid] == target)

          {

               return mid; // Target found at index mid

          }

          else if (arr[mid] < target)

          {

               left = mid + 1; // Search in the right half

          }

          else

          {

               right = mid - 1; // Search in the left half

          }

     }

     return -1; // Target not found

}

int main()

{

     int n;

     cin >> n;

     vector<int> arr(n);

     for (int i = 0; i < n; i++)

     {

          cin >> arr[i];

     }

     int target;

     cin >> target;

     sort(arr.begin(), arr.end());

     int result = binarySearch(arr, n, target);

     if (result != -1)

     {

          cout << "Element found at index " << result << endl;

     }

     else

     {

          cout << "Element not found in the array" << endl;

     }

     return 0;

}

## Sample Input 0:

5

1 5 2 4 3

2

## Sample Output 0:

Element found at index 1

## Sample Input 1:

5

1 5 2 4 3

7

## Sample Output 1:

Element not found in the array

# Problem number : 05

## Title : Write a program to find a given pattern form text using the pattern matching algorithm.

# Illustration of the problem :

The **Knuth-Morris-Pratt (KMP) algorithm** is an efficient string matching algorithm that finds occurrences of a **pattern** in a **text** in **O(n + m) time complexity**, where n is the length of the text and m is the length of the pattern. The primary advantage of the KMP algorithm over the brute-force approach is that it avoids unnecessary comparisons by using information from previous matches.

The key idea behind the KMP algorithm is to preprocess the pattern and construct a **Longest Prefix Suffix (LPS) array**. This array helps in avoiding redundant comparisons by determining how much the pattern should be shifted when a mismatch occurs.

## Time complexity :

> **Computing the LPS Array:** O(m)

> **Searching for the Pattern:** O(n)

> **Overall Complexity:** O(n + m)

# Algorithm :

**The KMP algorithm consists of two main steps:**

Step 1: Compute the Longest Prefix Suffix (LPS) Array

Step 2: Pattern Searching Using KMP

# Source Code :

## // this code is written by APURBO SHARMA (220604)

#include <iostream>

#include <vector>

using namespace std;

// Function to compute the Longest Prefix Suffix (LPS) array

void computeLPSArray(const string &pattern, vector<int> &lps)

{

    int m = pattern.length();

    int len = 0; // Length of the previous longest prefix suffix

    lps[0] = 0;  // lps[0] is always 0

    int i = 1;

    while (i < m)

    {

        if (pattern[i] == pattern[len])

        {

            len++;

            lps[i] = len;

            i++;

        }

        else

        {

            if (len != 0)

            {

                len = lps[len - 1]; // Reduce length to previous lps value

            }

            else

            {

                lps[i] = 0;

                i++;

            }

        }

    }

}

// Function to perform KMP pattern searching

void KMPSearch(const string &text, const string &pattern)

{

    int n = text.length();

    int m = pattern.length();

    // Create the LPS array

    vector<int> lps(m);

    computeLPSArray(pattern, lps);

    int i = 0; // Index for text

    int j = 0; // Index for pattern

    while (i < n)

    {

        if (pattern[j] == text[i])

        {

            i++;

            j++;

        }

        if (j == m)

        {

            cout << "Pattern found at index " << i - j << endl;

            j = lps[j - 1]; // Move to previous LPS position

        }

        else if (i < n && pattern[j] != text[i])

        {

            if (j != 0)

                j = lps[j - 1]; // Move j to previous LPS value

            else

                i++; // Move to the next character in text

        }

    }

}

int main()

{

    string text = "ababcababcabcab";

    string pattern = "abcab";

    KMPSearch(text, pattern);

    return 0;

}

## Sample Input 1:

## Text: "this is a simple example"

## Pattern: "simple"

## Sample Output 1:

Pattern found at index: 10

# Problem number : 06

## Title : Write a program to implement a queue data structure along with its typical operations.

# Illustration of the problem :

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. This means that the element added first will be removed first. A queue is similar to a real-world queue (e.g., people standing in line for a service).

**Operations in the Queue Class**

1. **enqueue(item)** – Adds an element to the end of the queue.
2. **dequeue()** – Removes and returns the front element of the queue.
3. **is\_empty()** – Checks whether the queue is empty.
4. **peek()** – Returns the front element without removing it.
5. **display()** – Displays all elements in the queue.

# Algorithm :

**Step 1: Initialize Queue**

* Create an empty list to store queue elements.

**Step 2: Enqueue Operation (Adding an element to the queue)**

* Append the new element to the list.
* **Code:** self.queue.append(item).

**Step 3: Dequeue Operation (Removing an element from the queue)**

* Check if the queue is **not empty**:
  + Remove and return the first element using self.queue.pop(0).
* If the queue is **empty**, return "Queue is empty".

**Step 4: Check if Queue is Empty**

* If len(self.queue) == 0, return True.
* Otherwise, return False.

**Step 5: Peek Operation (Viewing the front element without removal)**

* If the queue is **not empty**, return the first element.
* If the queue is **empty**, return "Queue is empty".

**Step 6: Display Queue**

* Return the list containing all queue elements.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

# Class definition for Queue

class Queue:

    def \_\_init\_\_(self):

        """Initialize an empty queue using a list"""

        self.queue = []

    def enqueue(self, item):

        """Add an element to the end of the queue"""

        self.queue.append(item)

    def dequeue(self):

        """Remove and return the front element of the queue"""

        if not self.is\_empty():  # Check if the queue is not empty

            return self.queue.pop(0)  # Remove and return the first element

        return "Queue is empty"  # Return a message if the queue is empty

    def is\_empty(self):

        """Check if the queue is empty"""

        return len(self.queue) == 0  # Returns True if queue is empty, else False

    def peek(self):

        """Return the front element without removing it"""

        if not self.is\_empty():

            return self.queue[0]  # Return first element

        return "Queue is empty"  # Return a message if the queue is empty

    def display(self):

        """Return the list representing the queue"""

        return self.queue

# Example usage

q = Queue()  # Create an empty queue

# Enqueue elements

q.enqueue(10)

q.enqueue(20)

q.enqueue(30)

# Display queue after enqueue operations

print("Queue after enqueue:", q.display())  # Output: [10, 20, 30]

# Dequeue an element

print("Dequeued element:", q.dequeue())  # Output: 10

# Display queue after dequeue operation

print("Queue after dequeue:", q.display())  # Output: [20, 30]

## Sample Input 1:

## q.enqueue(10)

## q.enqueue(20)

## q.enqueue(30)

## Sample Output 1:

Queue after enqueue: [10, 20, 30]

Dequeued element: 10

Queue after dequeue: [20, 30]

# Problem number : 07

## Title : Write a program to solve n queen’s problem using backtracking.

# Illustration of the problem :

The **N-Queens problem** is a classic backtracking problem where we need to place **N queens on an N × N chessboard** such that:

1. No two queens attack each other.
2. Queens **cannot be in the same row, column, or diagonal**.

Each solution is represented as an **array (solution vector)** where the **index represents the row** and the **value at that index represents the column** where the queen is placed.

# Algorithm :

**Step 1: Initialize the Board**

* Use a **1D array (board[n])** to store the queen positions.
* Each index represents a **row**, and the value at that index represents the **column** where the queen is placed.

**Step 2: Backtracking Function (solve)**

* Start placing queens **row by row**.
* For each row, try placing a queen in every column and check if it's a **safe position**.

**Step 3: Check Safety (is\_safe)**

For each new queen at board[row] = col, check:

1. **Same Column:** If another queen is already in the same column.
2. **Left Diagonal:** If another queen is in the same diagonal (board[i] - i == col - row).
3. **Right Diagonal:** If another queen is in the same diagonal (board[i] + i == col + row).

**Step 4: Recursive Placement**

* If a **valid placement** is found, move to the **next row** (solve(row + 1, board, solutions)).
* If row == n, a valid solution is found and added to solutions.

**Step 5: Backtrack**

* If no valid position is found for the current row, **undo the last move (backtrack)** and try a different column.

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <iostream>

#include <vector>

using namespace std;

// Function to check if placing a queen at (row, col) is safe

bool is\_safe(int row, int col, vector<int> &board)

{

   for (int i = 0; i < row; i++)

   {

      if (board[i] == col ||           // Same column

          board[i] - i == col - row || // Same main diagonal

          board[i] + i == col + row)

      { // Same anti-diagonal

         return false;

      }

   }

   return true;

}

// Recursive function to solve the N-Queens problem

void solve(int row, int n, vector<int> &board, vector<vector<int>> &solutions)

{

   if (row == n)

   {                              // If all queens are placed

      solutions.push\_back(board); // Store the solution

      return;

   }

   for (int col = 0; col < n; col++)

   {

      if (is\_safe(row, col, board))

      {                                       // Check if it's safe

         board[row] = col;                    // Place the queen

         solve(row + 1, n, board, solutions); // Move to the next row

         board[row] = -1;                     // Backtrack

      }

   }

}

// Function to solve N-Queens and return all solutions

vector<vector<int>> solve\_n\_queens(int n)

{

   vector<vector<int>> solutions; // Store all valid solutions

   vector<int> board(n, -1);      // Initialize board with -1 (no queen placed)

   solve(0, n, board, solutions); // Start solving from row 0

   return solutions;

}

// Function to print all solutions

void print\_solutions(vector<vector<int>> &solutions, int n)

{

   cout << "Number of solutions for " << n << "-Queens: " << solutions.size() << endl;

   for (auto solution : solutions)

   {

      cout << "Solution vector: ";

      for (int col : solution)

      {

         cout << col << " ";

      }

      cout << endl;

   }

}

// Main function

int main()

{

   int n;

   cout << "Enter the number of queens: ";

   cin >> n;

   vector<vector<int>> solutions = solve\_n\_queens(n);

   print\_solutions(solutions, n);

   return 0;

}

# Sample Input 0:

n = 5

# Sample Output 0:

Number of solutions for 5-Queens: 10

Solution vector: [0, 2, 4, 1, 3]

Solution vector: [0, 3, 1, 4, 2]

Solution vector: [1, 3, 0, 2, 4]

Solution vector: [1, 4, 2, 0, 3]

Solution vector: [2, 0, 3, 1, 4]

Solution vector: [2, 4, 1, 3, 0]

Solution vector: [3, 0, 2, 4, 1]

Solution vector: [3, 1, 4, 2, 0]

Solution vector: [4, 1, 3, 0, 2]

Solution vector: [4, 2, 0, 3, 1]

# Problem number : 08

## Title : Consider a set S = ( 5,10,12,13,15,18 ) and d = 30. Write a program to solve the sum of subset problem.

# Illustration of the problem :

Given a set **S = {5, 10, 12, 13, 15, 18}** and a target sum **d = 30**, we need to find all possible subsets of **S** whose sum equals **30**.

**Example Output:**

Possible subsets that sum to **30**:

* {5, 10, 15}
* {5, 12, 13}
* {12, 18}

This is solved using **backtracking**, where we explore different combinations of elements while ensuring the sum does not exceed **30**.

# Algorithm :

## Step-1 :

**Sort** the set S (optional for optimization).

## Step-2 :

Use **backtracking**:

* Start with an empty subset.
* **Include or exclude** elements recursively.
* Keep track of the **current sum**.
* **If sum == d (30), print the subset**.
* **If sum exceeds d, backtrack**.

## Step-3:

Repeat until all possibilities are explored.

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <bits/stdc++.h>

using namespace std;

// Function to find subsets with the given target sum

void sum\_of\_subsets(vector<int> &S, int target, vector<int> subset, int index)

{

   // Base Case: If the sum of subset equals target, print it

   int sum = 0;

   for (int num : subset)

      sum += num;

   if (sum == target)

   {

      cout << "Subset found: ";

      for (int num : subset)

         cout << num << " ";

      cout << endl;

      return;

   }

   // If sum exceeds target or all elements are considered, return

   if (sum > target || index == S.size())

      return;

   // Include the current element and recurse

   subset.push\_back(S[index]);

   sum\_of\_subsets(S, target, subset, index + 1);

   // Exclude the current element and recurse

   subset.pop\_back();

   sum\_of\_subsets(S, target, subset, index + 1);

}

int main()

{

   vector<int> S = {5, 10, 12, 13, 15, 18};

   int target\_sum = 30;

   cout << "Subsets with sum " << target\_sum << " are:" << endl;

   sum\_of\_subsets(S, target\_sum, {}, 0);

   return 0;

}

# Sample Input 0:

5, 10, 12, 13, 15, 18

# Sample Output 0:

Subsets with sum 30 are:

Subset found: [5, 10, 15]

Subset found: [5, 12, 13]

Subset found: [12, 18]

# Problem number : 09

## Title : 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 the number of items n = 4.

# Illustration of the problem :

**Knapsack Problem (0/1 Knapsack)**

* Given **n items**, each with:
  + **Weight** W[i]
  + **Profit/Value** P[i]
* A **knapsack** that can hold at most C weight.
* Find the **maximum profit** we can obtain by selecting some of the items **without exceeding the capacity**.

**Time & Space Complexity Analysis**

* **Time Complexity:** O(n \* C), where n is the number of items and C is the capacity.
* **Space Complexity:** O(n \* C) for the dp table.

# Algorithm :

**Step 1: Initialization**

* Create a **2D DP table (dp[n+1][C+1])** initialized with 0.
* dp[i][w] stores the **maximum profit** for first i items and capacity w.

**Step 2: Build DP Table**

* Iterate over **each item** i = 1 to n:
  + Iterate over **each weight capacity** w = 1 to C:
    - If W[i-1] ≤ w (item can be included):
      * **Include the item:** profit = P[i-1] + dp[i-1][w - W[i-1]]
      * **Exclude the item:** profit = dp[i-1][w]
      * Take the **maximum of these two**.
    - If W[i-1] > w (item too heavy), exclude it.

**Step 3: Return Result**

* The **final answer** is stored in dp[n][C].

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <iostream>

#include <vector>

using namespace std;

// Function to solve the 0/1 Knapsack Problem using Dynamic Programming

int knapsack(int C, vector<int>& W, vector<int>& P, int n) {

    // Create a DP table where dp[i][w] stores the max profit for first 'i' items and capacity 'w'

    vector<vector<int>> dp(n + 1, vector<int>(C + 1, 0));

    // Fill the DP table iteratively

    for (int i = 1; i <= n; i++) { // Iterate over each item

        for (int w = 1; w <= C; w++) { // Iterate over each weight capacity

            if (W[i - 1] <= w) // If the item can fit in the current capacity

                dp[i][w] = max(P[i - 1] + dp[i - 1][w - W[i - 1]], dp[i - 1][w]); // Include or exclude the item

            else

                dp[i][w] = dp[i - 1][w]; // If item weight exceeds capacity, exclude it

        }

    }

    return dp[n][C]; // Return the maximum profit for 'n' items and knapsack capacity 'C'

}

int main() {

    // Profit values for items

    vector<int> P = {15, 25, 13, 23};

    // Corresponding weight values for items

    vector<int> W = {2, 6, 12, 9};

    int C = 20;  // Knapsack capacity

    int n = P.size(); // Number of items

    // Call knapsack function and print the maximum profit

    cout << "Maximum Profit: " << knapsack(C, W, P, n) << endl;

    return 0;

}

# Sample Input 0:

vector<int> P = {15, 25, 13, 23}; // Profits of items

vector<int> W = {2, 6, 12, 9}; // Weights of items

int C = 20; // Knapsack capacity

# Sample Output 0:

Maximum Profit: 63

# Problem number : 10

## Title : Write a program to solve the Tower of Hanoi problem for the N disk.

# Illustration of the problem :

**The Tower of Hanoi is a classic problem that involves three pegs and a set of disks of different sizes. The objective is to move all the disks from one peg to another, following these rules:**

1. **Only one disk can be moved at a time.**
2. **A disk can only be placed on top of a larger disk or an empty peg.**
3. **All disks start on a single peg in increasing size order.**

**For N disks, the problem becomes exponentially complex. I’ll explain graphically, starting with the basic case for 3 disks, and how this process works for general N.**

**Time Complexity**: **O(2ⁿ)** (Exponential)

# Algorithm :

**Recursive Steps**

**Step-1 : Move (n-1) disks** from A → B using C as auxiliary.

**Step-2 : Move the largest disk (nth disk)** from A → C.

**Step-3 : Move (n-1) disks** from B → C using A as auxiliary.

# Source code :

## // this code is written by APURBO SHARMA (220604)

#include <iostream>

using namespace std;

int steps = 0; // Global variable to count steps

// Recursive function to solve Tower of Hanoi

void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod)

{

    // Base case: Move the last (smallest) disk directly

    if (n == 1)

    {

        cout << "Move disk 1 from " << from\_rod << " to " << to\_rod << endl;

        steps++; // Increment step counter

        return;

    }

    // Step 1: Move top n-1 disks from 'from\_rod' to 'aux\_rod' using 'to\_rod' as auxiliary

    towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

    // Step 2: Move the nth disk directly from 'from\_rod' to 'to\_rod'

    cout << "Move disk " << n << " from " << from\_rod << " to " << to\_rod << endl;

    steps++; // Increment step counter

    // Step 3: Move the n-1 disks from 'aux\_rod' to 'to\_rod' using 'from\_rod' as auxiliary

    towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

int main()

{

    int n;

    cout << "Enter number of disks: ";

    cin >> n;

    // Call Tower of Hanoi function (A = source, C = destination, B = auxiliary)

    towerOfHanoi(n, 'A', 'C', 'B');

    // Print total steps taken

    cout << "Total steps required: " << steps << endl;

    return 0;

}

# Sample Input 0:

Enter number of disks: 3

# Sample Output 0:

Enter number of disks: 3

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

Total steps required: 7