



# Problem No : 01

## Problem **Name :** Write a program to sort a linear array using the bubble sort algorithm.

#### **Theory :**

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It works by making multiple passes through the array until it is fully sorted. The largest elements gradually "bubble up" to their correct position at the end of the array in each pass, hence the name **Bubble Sort**.

### **Algorithm :**

1. Start with the first element.
2. Compare it with the next element.
3. If the first element is greater than the second, swap them.
4. Move to the next element and repeat.
5. The largest element will "bubble up" to the last position.
6. Repeat for the remaining elements until the entire array is sorted.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

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

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n - 1; i++) {

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

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

            if (arr[j] > arr[j + 1]) {

                // Swap arr[j] and arr[j+1]

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

            // Print array after each swap

            cout << "After swapping indices " << j << " and " << j + 1 << ": ";

            printArray(arr, n);

        }

        cout << "End of pass " << i + 1 << ": ";

        printArray(arr, n);

        cout << endl;

    }

}

int main() {

    int n;

    // Input the size of the array

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    // Input the array elements

    cout << "Enter the elements of the array: \n";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Sort the array using Bubble Sort

    cout << "Sorting process:\n";

    bubbleSort(arr, n);

    // Output the sorted array

    cout << "Final sorted array: ";

    printArray(arr, n);

    return 0;

}

**Sample Input & Output** :  
  
Enter the number of elements in the array: 5  
Enter the elements of the array: 6 5 2 8 4  
Sorting process:  
Pass 1:  
After swapping indices 0 and 1: 5 6 2 8 4  
After swapping indices 1 and 2: 5 2 6 8 4  
After swapping indices 2 and 3: 5 2 6 8 4  
After swapping indices 3 and 4: 5 2 6 4 8  
End of pass 1: 5 2 6 4 8  
  
  
Pass 2:  
After swapping indices 0 and 1: 2 5 6 4 8  
After swapping indices 1 and 2: 2 5 6 4 8  
After swapping indices 2 and 3: 2 5 4 6 8  
End of pass 2: 2 5 4 6 8  
Pass 3:  
After swapping indices 0 and 1: 2 5 4 6 8  
After swapping indices 1 and 2: 2 4 5 6 8  
End of pass 3: 2 4 5 6 8  
Pass 4:  
After swapping indices 0 and 1: 2 4 5 6 8  
End of pass 4: 2 4 5 6 8  
Final sorted array: 2 4 5 6 8

# Problem No : 02

**Problem** Name : **Write a program to find an element using a linear search algorithm.**

#### **Theory :**

Linear Search is a simple searching algorithm used to find the position of a target element in an array. It sequentially checks each element of the array until it finds the desired element or reaches the end of the array.

### **Algorithm :**

1. Take an array and a target element as input.
2. Traverse the array from the beginning.
3. If an element matches the target, return its index.
4. If the element is not found, return -1.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n)

{

    for (int i = 0; i < n - 1; i++)

    {

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

        {

            if (arr[j] > arr[j + 1])

            {

                // Swap arr[j] and arr[j+1]

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

int linearSearch(int arr[], int n, int key)

{

    for (int i = 0; i < n; i++)

    {

        if (arr[i] == key)

        {

            return i; // Return the index if found

        }

    }

    return -1; // Return -1 if not found

}

int main()

{

    int n;

    // Input the size of the array

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    // Input the array elements

    cout << "Enter the elements of the array: \n";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Sort the array using Bubble Sort

    bubbleSort(arr, n);

    // Output the sorted array

    cout << "Sorted array: \n";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    // Linear search for an element

    int key;

    cout << "Enter the element to search for: ";

    cin >> key;

    int result = linearSearch(arr, n, key);

    if (result != -1) {

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

    } else {

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

    }

    return 0;

}

**Sample Input & Output** :

Enter the number of elements in the array: 5  
Enter the elements of the array: 5 6 2 8 2  
Sorted array: 2 2 5 6 8  
Enter the element to search for: 5  
Element found at index: 3

# Problem No : 03

**Problem** Name : **Write a program to sort a linear array using the merge sort algorithm.**

#### **Theory :**

Merge Sort is a **divide-and-conquer** sorting algorithm that divides an array into smaller subarrays, sorts them, and then merges them back into a sorted array. It follows these steps:

1. **Divide**: Recursively split the array into two halves until each subarray contains only one element.
2. **Conquer**: Sort and merge the subarrays step by step.
3. **Combine**: Merge the sorted subarrays to form a fully sorted array.

### **Algorithm :**

1. If the array has one or zero elements, return (base case).
2. Find the middle index of the array.
3. Recursively divide the left and right halves.
4. Merge the sorted halves by comparing elements and arranging them in order.
5. Return the fully sorted array.

**Code in C++** :

#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Create temporary arrays

    int L[n1], R[n2];

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

    for (int i = 0; i < n1; i++)

        L[i] = arr[left + i];

    for (int i = 0; i < n2; i++)

        R[i] = arr[mid + 1 + i];

    // Merge the temporary arrays back into arr[left..right]

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    // Copy the remaining elements of L[], if any

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    // Copy the remaining elements of R[], if any

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

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

    if (left < right) {

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

        // Sort first and second halves

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves

        merge(arr, left, mid, right);

    }

}

int main() {

    int n;

    // Input the size of the array

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    // Input the array elements

    cout << "Enter the elements of the array: \n";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    // Sort the array using Merge Sort

    mergeSort(arr, 0, n - 1);

    // Output the sorted array

    cout << "Sorted array: \n";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    return 0;

}

**Sample Input & Output** :

Enter the number of elements in the array: 5  
Enter the elements of the array: 5 8 2 6 4  
Sorted array: 2 4 5 6 8

# Problem No : 04

**Problem** Name : **Write a program to find an element using the binary search algorithm.**

#### **Theory :**

Binary Search is a highly efficient algorithm for finding an element in a **sorted array** or **sorted list**. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues in the lower half, or if the value is greater, it continues in the upper half. This continues until the value is found or the interval is empty.

### **Algorithm :**

1. Initialize low = 0 and high = n - 1.
2. Find mid = (low + high) / 2.
3. If arr[mid] == target, return mid.
4. If arr[mid] > target, search in the left half (high = mid - 1).
5. If arr[mid] < target, search in the right half (low = mid + 1).
6. Repeat until low > high. If not found, return -1.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

int binarySearch(int arr[], int left, int right, int key) {

    while (left <= right) {

        int mid = (left + right) / 2;

        // Check if key is present at mid

        if (arr[mid] == key)

            return mid;

        // If key is greater, ignore the left half

        if (arr[mid] < key)

            left = mid + 1;

        // If key is smaller, ignore the right half

        else

            right = mid - 1;

    }

    return -1; // Key not found

}

int main() {

    int n;

    // Input the size of the array

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    // Input the array elements

    cout << "Enter the elements of the array: \n";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    sort(arr,arr+n);

     cout << "After Sorting the elements of the array: \n";

    for (int i = 0; i < n; i++) {

        cout<<arr[i]<<" ";

    }

    cout<<'\n';

    // Input the element to search for

    int key;

    cout << "Enter the element to search for: ";

    cin >> key;

    // Perform binary search

    int result = binarySearch(arr, 0, n - 1, key);

    if (result != -1) {

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

    } else {

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

    }

    return 0;

}

**Sample Input & Output** :

Enter the number of elements in the array: 5  
Enter the elements of the array: 4 2 8 6 3  
After Sorting the elements of the array: 2 3 4 6 8  
Enter the element to search for: 4  
Element found at index: 2

# Problem No : 05

**Problem** Name : **Write a program to find a given pattern form text using the pattern matching algorithm.**

#### **Theory :**

Pattern matching is the process of checking whether a given **pattern** (a string) exists within another string, called **text**. The simplest algorithm to perform pattern matching is the **Naive approach**, which checks all possible positions in the text where the pattern could match.

### **Algorithm :**

1. Start from the beginning of the text.
2. For each possible starting position of the text (from index 0 to n - m), compare the substring of length m with the pattern.
3. If all characters match, return the index.
4. If no match is found by the end of the text, return -1.

**Code in C++** **:**

#include <bits/stdc++.h>

using namespace std;

void PatternMatching(string &text,string &pattern) {

    int counter=0;

    int n = text.size();

    int m = pattern.size();

    for (int i = 0; i < n; i++) {

        int j = 0;

        if(text[i]==' ')continue;

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

            j++;

        }

        if (j == m) {

                counter++;

            cout << "Pattern found at index " << i<<" to "<<(i+m-1) << endl;

        }

    }

    cout<<"The number of matching pattern: "<<counter<<endl;

}

int main() {

    cout<<"Enter the text: "<<endl;

    string text;

    fflush(stdin);

    getline(cin,text);

    cout<<"Enter the pattern text:"<<endl;

    string pattern;

    cin>>pattern;

    PatternMatching(text, pattern);

    return 0;

}

**Sample Input & Output** :

Enter the text: You are good you are good  
Enter the pattern text: good  
Pattern found at index 8 to 11  
Pattern found at index 21 to 24  
The number of matching pattern: 2

# Problem No : 06

**Problem** Name : **Write a program to implement a queue data structure along with its typical operations.**

#### **Theory :**

A Queue is a linear data structure that follows the First In, First Out (FIFO) principle. In this structure, the element that is inserted first will be the first to be removed, similar to how a line works at a ticket counter. The two main operations in a queue are:

* **Enqueue**: Adds an element to the rear of the queue.
* **Dequeue**: Removes an element from the front of the queue.

### **Algorithm :**

1. **Enqueue**: Add element to rear.  
   If full, return error. Otherwise, increment rear.
2. **Dequeue**: Remove element from front.  
   If empty, return error. Otherwise, increment front.
3. **Front**: Return front element.
4. **IsEmpty**: Return true if size is 0.
5. **IsFull**: Return true if size equals capacity.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

// Node structure for the linked list

class Node {

    public:

    int data;

    Node\* next;

    Node(int value)

    {

        data=value;

        next=nullptr;

    }

};

// Queue class using linked list

class Queue {

private:

    Node\* front;

    Node\* rear;

public:

    Queue()

    {

        front=nullptr;

        rear=nullptr;

    }

    bool isEmpty() {

        return front == nullptr;

    }

    void enqueue(int value) {

        Node\* newNode = new Node(value);

        if (rear == nullptr) {

            front = rear = newNode;

        } else {

            rear->next = newNode;

            rear = newNode;

        }

        cout << value << " enqueued to the queue" << endl;

    }

    void dequeue() {

        if (isEmpty()) {

            cout << "Queue is empty.You Cannot dequeue." << endl;

            return;

        }

        Node\* temp = front;

        front = front->next;

        if (front == nullptr) {

            rear = nullptr;

        }

        cout << temp->data << " dequeued from the queue" << endl;

        delete temp;

    }

    int peek() {

        if (isEmpty()) {

            cout << "Queue is empty." << endl;

            return -1;

        }

        return front->data;

    }

};

int main() {

    Queue q;

    q.enqueue(10);

    q.enqueue(20);

    q.enqueue(30);

    cout << "Front element is: " << q.peek() << endl;

    q.dequeue();

    cout << "Front element after dequeue is: " << q.peek() << endl;

    q.dequeue();

    cout << "Front element after dequeue is: " << q.peek() << endl;

    q.dequeue();

    q.dequeue();

    return 0 ;

}

**Sample Input & Output** :

10 enqueued to the queue  
20 enqueued to the queue  
30 enqueued to the queue  
Front element is: 10  
10 dequeued from the queue  
Front element after dequeue is: 20  
20 dequeued from the queue  
Front element after dequeue is: 30  
30 dequeued from the queue  
Queue is empty. You Cannot dequeue.

# Problem No : 07

**Problem** Name : **Write a program to solve n queen’s problem using backtracking.**

#### **Theory :**

The N-Queens problem is a classic problem in computer science and mathematics. It asks how to place **N queens** on an **N x N** chessboard such that no two queens threaten each other. A queen can attack another queen if they share the same row, column, or diagonal.

### **Algorithm :**

1. Start placing queens one by one in different columns of the first row.
2. For each row, place the queen in a column and check for safety.
3. If safe, move to the next row. If not, backtrack to the previous row.
4. Repeat until all queens are placed or all possibilities are exhausted.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

bool isSafe(int board[], int row, int col, int N) {

    // Check if there's a queen in the same column or diagonals

    for (int i = 0; i < row; i++) {

        if (board[i] == col || abs(board[i] - col) == abs(i - row)) {

            return false;

        }

    }

    return true;

}

bool solveNQueens(int board[], int row, int N) {

    // If all queens are placed, return true

    if (row == N) {

        for (int i = 0; i < N; i++) {

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

                if (board[i] == j) {

                    cout << "Q ";  // Print queen at position

                } else {

                    cout << ". ";  // Empty space

                }

            }

            cout << endl;

        }

        return true;

    }

    // Try placing queen in each column

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

        if (isSafe(board, row, col, N)) {

            board[row] = col;  // Place queen at (row, col)

            if (solveNQueens(board, row + 1, N)) {  // Recur for next row

                return true;

            }

            board[row] = -1;  // Backtrack

        }

    }

    return false;  // No safe place for queen in this row

}

int main() {

    int N;

    cout << "Enter the size of the chessboard (N): ";

    cin >> N;

    if (N < 4) {

        cout << "No solution exists for N < 4." << endl;

        return 0;

    }

    int board[N];

    for (int i = 0; i < N; i++) {

        board[i] = -1;  // Initialize board with no queens placed

    }

    if (!solveNQueens(board, 0, N)) {

        cout << "No solution exists!" << endl;

    }

    return 0;

}

**Sample Input & Output** :

Enter the size of the chessboard (N): 4  
. Q . .  
. . . Q  
Q . . .  
. . Q .  
  
Enter the size of the chessboard (N): 3   
No solution exists for N < 4.

# Problem No : 08

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

#### **Theory :**

The **Sum of Subset Problem** is a classic example of **combinatorial optimization**. It involves finding a subset of a given set of numbers such that the sum of the numbers in the subset equals a given target value, d. The problem is NP-complete, meaning it is computationally expensive for large sets. However, there are several ways to approach this problem, with **backtracking** being one of the simplest methods.

### **Algorithm :**

1. Consider each element in the set S.
2. For each element, we have two choices:  
   Include the element in the subset.  
   Exclude the element from the subset.
3. If at any point, the sum of the current subset equals d, we have found a solution.
4. Backtrack if the current sum exceeds d or all elements have been considered.

**Code in C++** :

#include <iostream>

#include <vector>

using namespace std;

void printSubset(const vector<int>& subset) {

    // Print the subset

    for (int num : subset) {

        cout << num << " ";

    }

    cout << endl;

}

void findSubsetsWithSum(const vector<int>& S, int n, int d, vector<int>& currentSubset) {

    // Base case: if we've processed all elements

    if (d == 0) {

        printSubset(currentSubset);  // Print the current subset if the sum is 0

        return;

    }

    if (n == 0) {

        return;  // No more elements to check

    }

    // Exclude the current element and check the remaining subset

    findSubsetsWithSum(S, n - 1, d, currentSubset);

    // Include the current element if it doesn't exceed the target sum

    if (S[n - 1] <= d) {

        currentSubset.push\_back(S[n - 1]);

        findSubsetsWithSum(S, n - 1, d - S[n - 1], currentSubset);  // Recur with reduced sum

        currentSubset.pop\_back();  // Backtrack by removing the element

    }

}

int main() {

    // Given set S and target sum d

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

    int d = 30;

    int n = S.size();

    vector<int> currentSubset;  // To store the current subset being explored

    cout << "Subsets of the set that sum to " << d << " are: " << endl;

    findSubsetsWithSum(S, n, d, currentSubset);

    return 0;

}

**Sample Input & Output** :

Subsets of the set that sum to 30 are:  
13 12 5  
15 10 5  
18 12

# Problem No : 09

**Problem** 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 the number of items n = 4.**

#### **Theory :**

The **0/1 Knapsack Problem** is an optimization problem where given a set of items, each with a specific weight and profit, and a knapsack with a weight capacity, the goal is to determine the maximum profit that can be obtained by selecting a subset of items such that their total weight does not exceed the knapsack's capacity. The problem is typically solved using dynamic programming, where a 2D table is constructed to store solutions to subproblems. The value at each position of the table represents the maximum profit achievable with a certain number of items and a specific knapsack capacity. The solution is built up incrementally by considering each item and deciding whether to include it in the knapsack, based on whether it increases the total profit without exceeding the weight limit.

### **Algorithm :**

1. **Input**:
   * A set of n items with their weights w[i] and profits p[i].
   * The knapsack capacity C.
2. **Initialize the DP table**:
   * dp[i][w] where i ranges from 0 to n (items), and w ranges from 0 to C (capacity).
3. **Fill the DP table** using the recurrence relation described above.
4. **Return the result** stored in dp[n][C].

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

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

    // Create a DP table

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

    // Fill the DP table

    for (int i = 1; i <= n; ++i) {

        for (int w = 1; w <= C; ++w) {

            if (W[i-1] <= w) {

                dp[i][w] = max(dp[i-1][w], p[i-1] + dp[i-1][w-W[i-1]]);

            } else {

                dp[i][w] = dp[i-1][w];

            }

        }

    }

    // The result is stored in dp[n][C]

    return dp[n][C];

}

int main() {

    // Given profits, weights, and knapsack capacity

    vector<int> p = {15, 25, 13, 23};  // Profits

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

    int C = 20;  // Knapsack capacity

    int n = 4;    // Number of items

    // Call the knapsack function and print the result

    int maxProfit = knapsack(n, C, p, W);

    cout << "Maximum profit that can be obtained: " << maxProfit << endl;

    return 0;

}

**Sample Input & Output** :

Maximum profit that can be obtained: 63

# Problem No : 10

**Problem** Name : **Write a program to solve the Tower of Hanoi problem for the N disk.**

#### **Theory :**

The **Tower of Hanoi** is a classic problem in computer science and recursion. It involves three pegs and a number of disks of different sizes. Initially, all disks are stacked in decreasing size on one peg, known as the source peg. The goal is to move all disks from the source peg to a destination peg using the following rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one of the stacks and placing it on top of another stack.
3. A disk can only be placed on an empty peg or on top of a larger disk.

### **Algorithm :**

1. **If n == 1**, move the disk directly from source to destination.
2. **Otherwise**, recursively move n-1 disks from source to auxiliary peg.
3. Move the nth disk from source to destination.
4. Recursively move n-1 disks from auxiliary peg to destination peg.
5. Repeat until all disks are moved from the source to the destination peg.

**Code in C++** :

#include <bits/stdc++.h>

using namespace std;

// Recursive function to solve the Tower of Hanoi problem

void towerOfHanoi(int n, char source, char auxiliary, char destination) {

    // Base case: If there is only one disk, move it from source to destination

    if (n == 1) {

        cout << "Move disk 1 from " << source << " to " << destination << endl;

        return;

    }

    // Move the top n-1 disks from source to auxiliary peg

    towerOfHanoi(n - 1, source, destination, auxiliary);

    // Move the nth disk from source to destination

    cout << "Move disk " << n << " from " << source << " to " << destination << endl;

    // Move the n-1 disks from auxiliary to destination peg

    towerOfHanoi(n - 1, auxiliary, source, destination);

}

int main() {

    int n;  // Number of disks

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

    cin >> n;  // Taking user input for the number of disks

    // Call the recursive function to solve the Tower of Hanoi problem

    cout << "The moves involved are:" << endl;

    towerOfHanoi(n, 'A', 'B', 'C');  // A, B, C are the names of the pegs

    return 0;

}

**Sample Input & Output** :

Enter the number of disks: 2  
The moves involved are:  
Move disk 1 from A to B  
Move disk 2 from A to C  
Move disk 1 from B to C