

**Pabna University of Science and Technology, Pabna**

**Information and communication Engineering**

Course Name: Data Structure and Algorithm Sessional

Course Code: ICE-2202

Lab Report

SUBMITTED BY:

Md. Rifat Hossen

Roll: 220614

Session: 2021-2022

[Cite your source here.]

SUBMITTED TO:

[Md. Anwar Hossain](https://www.pust.ac.bd/academic/departments/dept_teachers/dept_teachers_profile/100072)

Professor

Department of Information and Communication Engineering, PUST

Submission Data: 28.02.2025

|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array using the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort a linear array using the merge sort algorithm. | |
| **4.** | |  | | --- | | Write a program to find an element using the binary search algorithm. | |
| **5.** | |  | | --- | | Write a program to find a given pattern from text using the pattern matching algorithm. | |
| **6.** | |  | | --- | | Write a program to implement a queue data structure along with its typical operations. | |
| **7.** | |  | | --- | | Write a program to solve **n** queen's problem using backtracking. | |
| **8.** | |  | | --- | | Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset problem. | |
| **9.** | |  | | --- | | 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. | |

**Index**

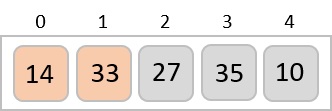
**Problem No -01**

**Title**: Write a program to sort an array using bubble sort algorithm.

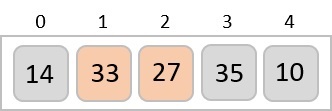
**Theory**: We take an unsorted array for our Example. Bubble sort takes O(n2) times.

14 33 27 35 10

Bubble sort starts with every first two elements comparing them to check which one is greater.



In this case, Value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller then 33 and these two value must be swapped.

Similarly, we know that we know that 35 is greater than 33, so it’s also swapped. And finally, 10 is smaller than all other value, so it also compares with all value of way so finally we find sorted array.

**Algorithm:**

1. Repeat step 2 and 3 for K=1 to N-1
2. Set PTR=1
3. Repeat while PTR<=N-K:
4. If DATA[PTR] > DATR[PTR+1] then,

Interchange DATA[PTR] and DATA[PTR+1]

1. Set PTR = =PTR+1;
2. Exist.

**Source Code:**

**#include <bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int n;**

**cin >> n;**

**int arr[n];**

**for (int i = 0; i < n; i++)**

**{**

**cin >> arr[i];**

**}**

**for (int i = 1; i < n; i++)**

**{**

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

**{**

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

**{**

**swap(arr[j], arr[j + 1]);**

**}**

**}**

**}**

**for (int i = 0; i < n; i++)**

**{**

**cout << arr[i] << " ";**

**}**

**return 0;**

**}**

**Input:**

14 33 27 35 10

**Output:**

10 14 27 33 35

**Problem -02:**

**Title:** Write a program to sort an array using Binary Search algorithm.

**Theory:** Suppose data is an array which is sorted in increasing numerical order or equivalently or alphabetically. Then there is an extremely efficient Searching Algorithm called binary search. Which can be used to find the location LOC of a given ITEM of Information in data.

**Algorithm: (Binary search) Binary (Array, Size, ITEM):**

1. Initialize **Low** and **High** pointer.
2. While **low<=high** :

* Calculate **mid = (low+high)/2**
* **If(ar[mid]==ITEM):**

Return it **index**.

* **If(ar[mid]<ITEM):**

**Low=mid+1**

* Else :

**High=mid -1**

1. Exist.

**Source code:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

    cin >> n;

    int ar[n];

    for (int i = 0; i < n; i++)

    {

        cin >> ar[i];

    }

    sort(ar, ar + n);

    int key;

    cin >> key;

    int l = 0;

    int r = n - 1;

    bool flag = false;

    while (l <= r)

    {

        int mid = (l + r) / 2;

        if (ar[mid] == key)

        {

            cout << ar[mid] << endl;

            flag = true;

            break;

        }

        else if (ar[mid] < key)

        {

            l = mid + 1;

        }

        else

        {

            r = mid - 1;

        }

    }

    if (!flag)

    {

        cout << "Not found" << endl;

    }

    return 0;

}

Input:

2 3 5 7 11 13 17 19 23 29

7

Output:

Index – 3

**Problem -03:**

**Title:** Write a program to sort an array using Linear search algorithm.

**Theory:** Linear search is a simple and straightforward searching technique. It is used to find the position of a target in list or array. The algorithm checks each element in the list sequentially util the desired element is found or the end of the list is reached.

**Algorithm: Linear Search (Array, ITEM):**

1. For i=0 to size of array:
2. Compare the **current element** with the **target.**
3. If the **current element** matches the **target**:

Return its **index**(position)

1. If loop end and **no match** is found

Return “**Not found”.**

1. Exist.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

    cin >> n;

    int ar[n];

    for (int i = 0; i < n; i++)

    {

        cin >> ar[i];

    }

    int key;

    cin >> key;

    bool found = false;

    for (int i = 0; i < n; i++)

    {

        if (ar[i] == key)

        {

            cout << i << endl;

            found = true;

            break;

        }

    }

    if (!found)

    {

        cout << "-1" << endl;

    }

    return 0;

}

**Input:**

5 3 7 10 15

10

**Output:**

Index: 3

**Problem No - 04**

**Titel**: Write a program to sort array using marge sort algorithm.

**Theory:**

Suppose an array a with n element A [1] ……………A[n] is in memory. The merge sort algorithm which sorts a will first be describe by means of a specific example.

Example: Suppose the array a contain 14 elements as follows:

12 23 45 36 24 25 63 56

The merge sort algorithm will start at the beginning of the array and merge pairs of sorted sub array as follows:

Step 1: merge each pair of elements it obtains the following list of sorted pairs:

12 23 45 36 24 25 63 56

Step 2: merge each pair of elements it obtains the following sorted algorithm:

12 23 45 36 24 25 63 56

Step 3: merge each pair of elements it obtains the following sorted algorithm:

12 23 36 45 24 25 56 63

Step 4: merge each pair of elements it obtains the following sorted algorithm:

12 23 24 45 56 63

**Algorithm: MEARGE sort (Array, N)**

1. Set L=1
2. Repeat step 3 to 6 while L<N:
3. Call MERGE(A,N,L,B)
4. Call MERGE(B,N,2\*L,A)
5. Set L= 4\*L

[End of step 2 loop]

1. Exist.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

void marge(int ar[], int l, int mid, int r)

{

int n1 = mid - l + 1;

int n2 = r - mid;

vector<int> L(n1), R(n2);

// copy the element;

for (int i = 0; i < n1; i++)

{

L[i] = ar[l + i];

}

for (int j = 0; j < n2; j++)

{

R[j] = ar[mid + 1 + j];

}

int i = 0, j = 0, k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

ar[k] = L[i];

i++;

}

else

{

ar[k] = R[j];

j++;

}

k++;

}

// copy the remaing element of L;

while (i < n1)

{

ar[k] = L[i];

i++;

k++;

}

while (j < n2)

{

ar[k] = R[j];

j++;

k++;

}

}

void margsort(int ar[], int l, int r)

{

if (l >= r)

{

return;

}

int mid = (l + r) / 2;

margsort(ar, l, mid);

margsort(ar, mid + 1, r);

marge(ar, l, mid, r);

}

void pritnvector(int ar[])

{

for (int i = 0; i < 6; i++)

{

cout << ar[i] << " ";

}

cout << endl;

}

int main()

{

int n = 6;

int ar[n] = {12, 11, 13, 5, 6, 7};

pritnvector(ar);

margsort(ar, 0, n - 1);

pritnvector(ar);

return 0;

}

**Input:**

12 23 45 36 24 25 63 56

**Output:**

12 23 24 45 56 63

**Problem No – 05**

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

**Theory:** The challenge of finding a substring, also known as a pattern, inside a longer string, also known as the text, is known as pattern matching. It is essential to numerous applications, including bioinformatics, data mining, and text searching. For example, ‘**’ABABDABACDABABCABAB’’** find the pattern **‘ABABCABAB’.**

**Step 1:** Initialize the lps array with zeros:  
lps = [0, 0, 0, 0, 0, 0, 0, 0, 0]  
Here, lps[i] will store the length of the longest proper prefix which is also a suffix for the pattern up to position i.

**Step 2:** We start building the lps array by comparing characters in the pattern.

* **i = 1:** Compare pattern[1] (which is B) with pattern[0] (which is A). They don't match, so lps[1] = 0.
* **i = 2:** Compare pattern[2] (which is A) with pattern[0] (which is A). They match, so lps[2] = 1.
* **i = 3:** Compare pattern[3] (which is B) with pattern[1] (which is B). They match, so lps[3] = 2.
* **i = 4:** Compare pattern[4] (which is C) with pattern[2] (which is A). They don't match, so lps[4] = 0.
* **i = 5:** Compare pattern[5] (which is A) with pattern[0] (which is A). They match, so lps[5] = 1.
* **i = 6:** Compare pattern[6] (which is B) with pattern[1] (which is B). They match, so lps[6] = 2.
* **i = 7:** Compare pattern[7] (which is A) with pattern[2] (which is A). They match, so lps[7] = 3.
* **i = 8:** Compare pattern[8] (which is B) with pattern[3] (which is B). They match, so lps[8] = 4.

**Algorithm:**

1. Initially calculate the hash value of the pattern P.
2. Start iterating from the start of the string:
3. Calculate the hash value of the current substring having length m.
4. If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
5. If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
6. Return the starting indices as the required answer.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

void patternMaching(string text, string pattern)

{

    int n = text.length();

    int m = pattern.length();

    bool found = false;

    // slide the pattern over the text

    for (int i = 0; i <= n - m; i++)

    {

        int j;

        for (j = 0; j < m; j++)

        {

            if (text[i + j] != pattern[j])

            {

                break;

            }

        }

        if (j == m)

        {

            found = true;

            cout << "pattern found at index : " << i << endl;

        }

    }

    if (!found)

    {

        cout << "pattern not found in the text" << endl;

    }

}

int main()

{

    string text, pattern;

    getline(cin, text);

    getline(cin, pattern);

    patternMaching(text, pattern);

    return 0;

}

**Input: ABABDABACDABABCABAB**

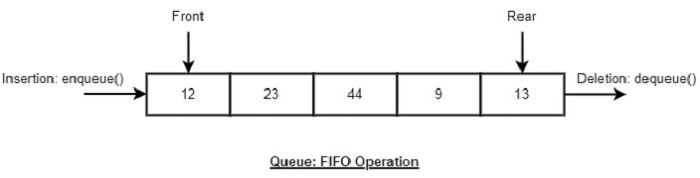
**Target: ABABCABAB**

**Output: True.**

**Problem No: 06**

**Title: Write a program to implement a queue data structure along with its typical operation**

**Theory:** A queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.



**Algorithm:**

* 1. START
  2. Check if the queue is full.
  3. If the queue is full, produce overflow error and exit.
  4. If the queue is not full, increment rear pointer to point the next empty space. 5. Add data element to the queue location, where the rear is pointing.
  5. return success.
  6. END

**Source code:**

**#include <bits/stdc++.h>**

**using namespace std;**

**class myqueuse**

**{**

**public:**

**list<int> l;**

**void push(int val)**

**{**

**l.push\_back(val);**

**}**

**void pop()**

**{**

**l.pop\_front();**

**}**

**int fron()**

**{**

**return l.front();**

**}**

**int size()**

**{**

**return l.size();**

**}**

**bool empty()**

**{**

**if (l.size() == 0)**

**{**

**return true;**

**}**

**else**

**{**

**return false;**

**}**

**}**

**};**

**int main()**

**{**

**myqueuse q;**

**int n;**

**cin >> n;**

**for (int i = 0; i, i < n; i++)**

**{**

**int x;**

**cin >> x;**

**q.push(x);**

**}**

**while (!q.empty())**

**{**

**cout << q.fron();**

**q.pop();**

**}**

**return 0;**

**}**

**Input: 5 10 20 30 40 50**

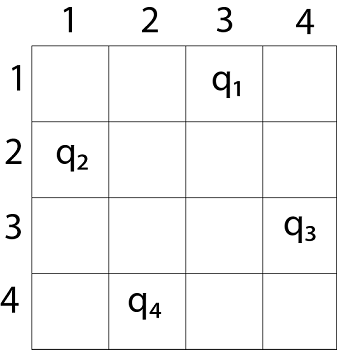
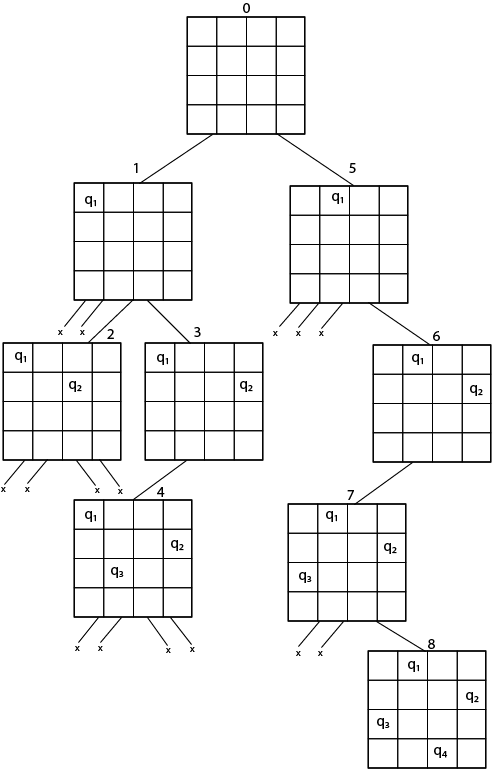
**Output: 10 20 30 40 50**

**Problem No: 07**

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

**Theory:** On a n × n chessboard, the N-Queens problem is to arrange n-queens so that no queens attack one another by being in the same row, column, or diagonal.   
  
It is evident that there is no solution for n = 2 and n = 3, but there is a straightforward solution for n = 1. In order to construct the n-queens problem, we will first investigate the four-queens problem.

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

**** 

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x1, x2, x3, x4) where xi represents the column on which queen "qi" is placed

**Algorithm:**

* Start in the leftmost column
* If all queens are placed return true
* Try all rows in the current column. Do the following for every row.
  + If the queen can be placed safely in this row
    - Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    - If placing the queen in [row, column] leads to a solution then return true.
    - If placing queen doesn’t lead to a solution then unmark this [row, column] then backtrack and try other rows.
  + If all rows have been tried and valid solution is not found return false to trigger backtracking.

**Source code:**

**#include <iostream>**

**#define N 8**

**using namespace std;**

**void printSolution(int board[N][N])**

**{**

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

**{**

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

**if (board[i][j])**

**cout << "Q ";**

**else**

**cout << ". ";**

**cout << "\n";**

**}**

**}**

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

**{**

**int i, j;**

**// Check this row on left side**

**for (i = 0; i < col; i++)**

**if (board[row][i])**

**return false;**

**// Check upper diagonal on left side**

**for (i = row, j = col; i >= 0 && j >= 0; i--, j--)**

**if (board[i][j])**

**return false;**

**// Check lower diagonal on left side**

**for (i = row, j = col; j >= 0 && i < N; i++, j--)**

**if (board[i][j])**

**return false;**

**return true;**

**}**

**bool solveNQUtil(int board[N][N], int col)**

**{**

**if (col >= N)**

**return true**

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

**{**

**if (isSafe(board, i, col))**

**{**

**board[i][col] = 1**

**if (solveNQUtil(board, col + 1))**

**return true;**

**board[i][col] = 0; // BACKTRACK**

**}**

**}**

**return false;**

**}**

**bool solveNQ()**

**{**

**int board[N][N] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};**

**if (solveNQUtil(board, 0) == false)**

**{**

**cout << "Solution does not exist";**

**return false;**

**}**

**printSolution(board);**

**return true;**

**}**

**int main()**

**{**

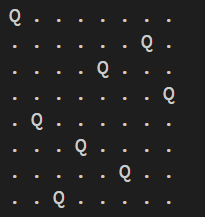
**solveNQ();**

**return 0;**

**}**

**Input: 4**

**Output:**

****

**Problem: 08**

**Title: Consider a set S= {5,10,12,13,15,18} and a target sum d=30d = 30d=30, find subsets whose elements sum up to 30.**

**Theory:** One significant NP-hard problem in combinatorial optimization is the set covering problem. Finding the least number of sets that contain (cover) a specific collection of objects is the goal of the set covering problem.   
Given a collection S of m sets and a set of elements 1, 2,..., n (referred to as the universe), the set cover problem is to determine the smallest sub-collection of S whose union matches the universe. Consider the group of sets S = 1, 2, 3, 2, 4, 3, and 4 with the universe U = 1, 2, 3, 4, 5. Without a doubt, the merger of S is U. However, all elements may be accommodated in the following smaller number of sets: 1, 2, 3, and 4.

**Algorithm:**

1. Use backtracking to explore all possible subsets.
2. Maintain a current subset and track the sum.
3. If the current sum matches the target, print the subset.
4. Backtrack to explore other subsets.

**Source Code:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**void findSubsets(vector<int>& set, vector<int>& subset, int index, int currentSum, int targetSum) {**

**if (currentSum == targetSum) {**

**// Print the subset**

**cout << "Subset: ";**

**for (int num : subset) {**

**cout << num << " ";**

**}**

**cout << endl;**

**return;**

**}**

**if (index >= set.size() || currentSum > targetSum) return;**

**// Include the current element in the subset**

**subset.push\_back(set[index]);**

**findSubsets(set, subset, index + 1, currentSum + set[index], targetSum);**

**// Exclude the current element (backtrack)**

**subset.pop\_back();**

**findSubsets(set, subset, index + 1, currentSum, targetSum);**

**}**

**int main() {**

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

**int targetSum = 30;**

**vector<int> subset;**

**cout << "Subsets that sum up to " << targetSum << ":" << endl;**

**findSubsets(set, subset, 0, 0, targetSum);**

**return 0;**

**}**

**Input:** S= {5,10,12,13,15,18} d=30

**Output:** 0, 1, 4.

**Problem No: 09**

**Title: 0/1 Knapsack Problem using Dynamic Programming in C++**

**Theory:** Given a bag with a capacity of W, meaning it can store a maximum of W weight, and N objects, each of which has a weight and profit connected with it. The goal is to pack the things in the bag so that the total profit from them is as high as it can be.  
Enter N = 3, W = 4, weight[] = {4, 5, 1}, and profit[] = {1, 2, 3}.   
Results: 3   
Two of the objects weigh less than or equal to four. The potential profit is one if we choose the item with weight four. Additionally, the potential profit is three if we choose the object with weight 1. Thus, three is the highest profit that may be made. Be aware that we are unable to enter 4**.**

**Algorithm**

1. Define a 2D DP table dp[n+1][capacity+1] where dp[i][j] stores the maximum profit for the first i items with knapsack capacity j.
2. Initialize the first row and column to 0 (no profit without items or with zero capacity).
3. For each item, decide whether to include or exclude it based on weight and profit:
   * Include: Profit from current item + optimal solution for remaining capacity.
   * Exclude: Optimal solution without the current item.
4. Return dp[n][capacity] as the solution.

**Source code:**

#include <iostream>

#include <vector>

using namespace std;

int knapsack(int capacity, vector<int>& weights, vector<int>& profits, int n) {

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

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

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

if (weights[i - 1] <= w) {

dp[i][w] = max(profits[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

} else {

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

}

}

}

return dp[n][capacity];

}

int main() {

vector<int> profits = {15, 25, 13, 23};

vector<int> weights = {2, 6, 12, 9};

int capacity = 20;

int n = profits.size();

int maxProfit = knapsack(capacity, weights, profits, n);

cout << "Maximum profit: " << maxProfit << endl;

return 0;

}

**Input:** N = 3, W = 4, profit [] = {1, 2, 3}, weight [] = {4, 5, 1}

**Output:** 3