**INFORMATION AND COMMUNICATION ENGINEERING**

**Course Name: Data Structure and Algorithm Sessional  
Course Code: ICE-2202**

**Submitted To,  
Md. Anwar Hossain**

**Professor  
Department of Information and Communication Engineering**

**Pabna University of Science and Technology**

**Submitted By**, **Nafis Fuad Rudro  
Roll: 220621  
Session: 2021-22**

**Department of Information and Communication Engineering**

**Pabna University of Science and Technology**

Faculty of Engineering and Technology  
 Pabna University of Science and Technology

**INDEX**

|  |  |
| --- | --- |
| **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. | |

# Lab Report 1: Write a program to sort a linear array using the bubble sort algorithm

Theory:

Bubble sort is one of the simplest sorting algorithms that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. It has an average and worst-case time complexity of O(n²), making it inefficient for large datasets. Despite its inefficiency, it is widely used in educational purposes to introduce sorting concepts.

## Problem Statement:

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

## Algorithm:

1. Start by comparing adjacent elements in the array.  
2. If the first element is greater than the second, swap them.  
3. Move to the next pair and repeat the comparison and swapping.  
4. Repeat the process for all elements in the array.  
5. Continue the passes through the array until no swaps are needed, meaning the array is sorted.  
6. Print the sorted array.

## C++ Code Implementation:

```cpp  
#include <iostream>  
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], arr[j + 1]);  
 }  
 }  
 }  
}  
int main() {  
 int arr[] = {64, 34, 25, 12, 22, 11, 90};  
 int n = sizeof(arr)/sizeof(arr[0]);  
 bubbleSort(arr, n);  
 cout << "Sorted array: ";   
 for (int i = 0; i < n; i++)  
 cout << arr[i] << " ";   
 return 0;  
}  
```

## Sample Input & Output:

Input: [64, 34, 25, 12, 22, 11, 90]  
Output: [11, 12, 22, 25, 34, 64, 90]

# Lab Report 2: Write a program to find an element using a linear search algorithm.

Linear search is a simple searching algorithm that sequentially checks each element in an array until the desired value is found or the end of the array is reached. It has a time complexity of O(n), making it inefficient for large datasets but useful for small or unsorted data.

## Problem Statement:

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

## Algorithm:

1. Start from the first element of the array.  
2. Compare the element with the target value.  
3. If a match is found, return the index of the element.  
4. If no match is found, continue to the next element.  
5. If the end of the array is reached, return -1 (element not found).

## C++ Code Implementation:

```cpp  
#include <iostream>  
using namespace std;  
int linearSearch(int arr[], int n, int x) {  
 for (int i = 0; i < n; i++) {  
 if (arr[i] == x)  
 return i;  
 }  
 return -1;  
}  
int main() {  
 int arr[] = {10, 20, 30, 40, 50};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 int x = 30;  
 int result = linearSearch(arr, n, x);  
 if (result == -1)  
 cout << "Element not found";  
 else  
 cout << "Element found at index " << result;  
 return 0;  
}  
```

## Sample Input & Output:

Input: [10, 20, 30, 40, 50], x = 30  
Output: Element found at index 2

# Lab Report 3: Write a program to sort a linear array using the merge sort algorithm.

Theory:

Merge Sort is a divide-and-conquer sorting algorithm that splits an array into halves, sorts them recursively, and then merges them back together. It has a time complexity of O(n log n), making it efficient for large datasets.

## Problem Statement:

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

## Algorithm:

1. Divide the array into two halves.  
2. Recursively sort each half.  
3. Merge the two sorted halves back together.

## C++ Code Implementation:

```cpp  
#include <iostream>  
using namespace std;  
void merge(int arr[], int left, int mid, int right) {  
 int n1 = mid - left + 1, n2 = right - mid;  
 int L[n1], R[n2];  
 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];  
 int i = 0, j = 0, k = left;  
 while (i < n1 && j < n2) {  
 if (L[i] <= R[j]) arr[k++] = L[i++];  
 else arr[k++] = R[j++];  
 }  
 while (i < n1) arr[k++] = L[i++];  
 while (j < n2) arr[k++] = R[j++];  
}  
void mergeSort(int arr[], int left, int right) {  
 if (left < right) {  
 int mid = left + (right - left) / 2;  
 mergeSort(arr, left, mid);  
 mergeSort(arr, mid + 1, right);  
 merge(arr, left, mid, right);  
 }  
}  
int main() {  
 int arr[] = {38, 27, 43, 3, 9, 82, 10};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 mergeSort(arr, 0, n - 1);  
 cout << "Sorted array: ";   
 for (int i = 0; i < n; i++)  
 cout << arr[i] << " ";   
 return 0;  
}  
```

## Sample Input & Output:

Input: [38, 27, 43, 3, 9, 82, 10]  
Output: [3, 9, 10, 27, 38, 43, 82]

**Lab Report 4:** Write a program to find an element using the binary search algorithm.

**Theory:**

Binary search is an efficient searching algorithm used for **sorted arrays**. It follows a **divide-and-conquer** approach by repeatedly dividing the array into halves and searching in the relevant half. It has a time complexity of **O(log n)**.

**Problem Statement:**

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

**Algorithm:**

1. Set the low and high pointers to the start and end of the array.
2. Find the middle element.
3. If the middle element matches the target, return its index.
4. If the target is smaller, search in the left half.
5. If the target is larger, search in the right half.
6. Repeat until the element is found or the search space is exhausted.

Code:

#include <iostream>

using namespace std;

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

while (left <= right) {

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

if (arr[mid] == x) return mid;

if (arr[mid] < x) left = mid + 1;

else right = mid - 1;

}

return -1;

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

int n = sizeof(arr) / sizeof(arr[0]);

int x = 30;

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

if (result == -1)

cout << "Element not found";

else

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

return 0;

}

**Sample Input & Output:**

**Input:** [10, 20, 30, 40, 50], x = 30  
**Output:** Element found at index 2

**Lab report 5:** Write a program to find a given pattern from text using the pattern matching algorithm

**Theory**

The Pattern Search Algorithm is a type of direct search method. It iteratively explores the search space using a set of predefined steps or patterns to generate new candidate solutions. The algorithm evaluates these solutions based on an objective function and moves to the best candidate point.

Unlike gradient-based methods, which rely on first and second derivatives of the objective function, the pattern search method evaluates function values at points within a search pattern. This makes it effective in scenarios where the function is not smooth or has discontinuities.

**Objective**

The objectives of this experiment are:

1. To implement the Pattern Search Algorithm in C++.
2. To analyze its performance using various test functions.
3. To explore its efficiency and convergence for different types of optimization problems.
4. To assess the algorithm’s ability to minimize the objective function.

### ****Algorithm (Steps)****

The steps of the Pattern Search Algorithm are as follows:

**Step 1:** Initialize the starting point x0x\_0x0​, the step size δ\deltaδ, and the maximum number of iterations NNN.

**Step 2:** Evaluate the objective function at the current point f(x0)f(x\_0)f(x0​).

**Step 3:** Generate a set of candidate points by perturbing the current point in each direction, using the step size δ\deltaδ.

**Step 4:** Evaluate the objective function at each candidate point.

**Step 5:** If a candidate point provides a better function value (lower value), update the current point to the best candidate.

**Step 6:** If no improvement occurs, reduce the step size δ\deltaδ and repeat the process.

**Step 7:** Stop when the step size becomes sufficiently small or the maximum number of iterations is reached.

Code Implementation (C++):

#include <iostream>

#include <cstring>

using namespace std;

 int main() {

    char text[1000], pattern[1000];

cout << "Enter the text: ";

    cin.getline(text, sizeof(text));

  cout << "Enter the pattern: ";

    cin.getline(pattern, sizeof(pattern));

  int textLen = strlen(text);

    int patternLen = strlen(pattern);

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

        int j = 0;

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

{

      j++;

     }

   if (j == patternLen) {

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

return 0;

        }

    }

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

  return 0;

}

Input/Output:

Enter the text: hello world

Enter the pattern: wor

Pattern found at index 6

**Lab report 6:** Write a program to implement a queue data structure along with its typical operations

Theory

A **Queue** is a fundamental data structure that follows the **First-In-First-Out (FIFO)** principle. In this structure, elements are inserted at the rear and removed from the front. This behavior is analogous to a queue in real life, such as people waiting in line at a service counter.

Queues are widely used in computer science for scheduling tasks, buffering data, and handling asynchronous events. Examples of queue applications include managing processes in operating systems, network traffic handling, and data processing.

**Objective:**

To implement a queue data structure in C++ and demonstrate its typical operations like enqueue, dequeue, peek, isEmpty, and isFull.

**Algorithm:**

1. **Enqueue Operation:** Insert the element at the rear of the queue, if the queue is not full.
2. **Dequeue Operation:** Remove the front element of the queue, if the queue is not empty.
3. **Peek Operation:** Retrieve the front element without removing it, if the queue is not empty.
4. **IsEmpty Operation:** Return true if the queue is empty (front == -1).
5. **IsFull Operation:** Return true if the queue is full (rear == MAX - 1).

Code:

#include <iostream>

using namespace std;

#define MAX 5

int queue[MAX], front = -1, rear = -1;

int main() {

int choice, value;

while (1) {

cout << "\nQueue Operations:\n";

cout << "1. Enqueue\n";

cout << "2. Dequeue\n";

cout << "3. Display\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

if (rear == MAX - 1) {

cout << "Queue is full! Cannot enqueue.\n";

} else {

if (front == -1) {

front = 0;

}

cout << "Enter the value to enqueue: ";

cin >> value;

rear++;

queue[rear] = value;

cout << value << " enqueued to the queue.\n";

cout << "Current front: " << front << ", rear: " << rear << endl;

}

break;

case 2:

if (front == -1 || front > rear) {

cout << "Queue is empty! Cannot dequeue.\n";

} else {

cout << "Dequeued value: " << queue[front] << endl;

front++;

cout << "Current front: " << front << ", rear: " << rear << endl;

if (front > rear) {

front = rear = -1;

cout << "Queue is now empty.\n";

}

}

break;

case 3:

if (front == -1 || front > rear) {

cout << "Queue is empty!\n";

} else {

cout << "Queue elements are: ";

for (int i = front; i <= rear; i++) {

cout << queue[i] << " ";

}

cout << endl;

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

cout << "Invalid choice! Try again.\n";

}

if (front == -1 || front > rear) {

cout << "Queue is empty.\n";

} else {

cout << "Current queue: ";

for (int i = front; i <= rear; i++) {

cout << queue[i] << " ";

}

cout << endl;

}

}

return 0;

}

Output :

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 3

3 enqueued to the queue.

Current front: 0, rear: 0

Current queue: 3

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 4

4 enqueued to the queue.

Current front: 0, rear:

1 Current queue: 3 4

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements are: 3 4

Current queue: 3 4

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit Enter your choice:

4 Exiting...

**Lab Report 7:** Write a program to solve **n** queen's problem using backtracking.

**Theory**

The N-Queens problem is a classic combinatorial problem that challenges one to place N chess queens on an N×N chessboard such that no two queens threaten each other. This report demonstrates how the problem can be efficiently solved using the backtracking algorithm. The lab covers the theoretical background, objective, algorithm, C++ implementation, and evaluation of the solution. The solution successfully places N queens on a board of any size while ensuring no two queens threaten each other.

**3. Objective**

Implement the N-Queens problem solution using the backtracking technique.

1. Demonstrate the effectiveness of the backtracking algorithm in finding solutions to the N-Queens problem for any N.
2. Analyze the time complexity and correctness of the algorithm.

**4. Algorithm**

The steps of the backtracking algorithm for solving the N-Queens problem are as follows:

1. Start from the first row.
2. Try placing a queen in each column of the current row.
   * If placing the queen in a column does not lead to a conflict with any previously placed queens, mark the current position as valid and move to the next row.
3. If placing a queen in the current row and column results in a valid configuration, proceed to place queens in the next row.
4. If placing a queen results in a conflict, backtrack by removing the last placed queen and try the next column in the current row.
5. Repeat steps 2-4 until all queens are placed on the board or all possibilities are exhausted.
6. If all queens are placed successfully, print the solution.
7. If there is no valid position for a queen in the current row, backtrack to the previous row.

Code:

#include <iostream>

using namespace std;

#define N 4

void printSolution(int placed[]) {

static int solutionCount = 0;

cout << "\nSolution " << ++solutionCount << ":\n";

for (int i = 0; i < N; i++, cout << "\n")

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

cout << (placed[i] == j ? 'Q' : '.') << " ";

}

bool isSafe(int placed[], int row, int col) {

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

if (placed[prev] == col ||

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

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

return false;

}

}

return true;

}

void solveNQueens(int placed[], int row) {

if (row == N) {

printSolution(placed);

return;

}

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

if (isSafe(placed, row, col)) {

placed[row] = col;

solveNQueens(placed, row + 1);

}

}

}

int main() {

int placed[N] = {-1};

solveNQueens(placed, 0);

return 0;

}

Output:

Solution 1:

. Q . .

. . . Q

Q . . .

. . Q .

Solution 2:

. . Q .

Q . . .

. . . Q

. Q . .

**Lab Report 8: 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 classical problem in computer science and combinatorial optimization. It involves determining whether there is a subset of a set SSS whose sum equals a target value ddd. This problem can be solved using various algorithms, such as:

* **Brute-force approach:** Evaluate all subsets of the set SSS.
* **Dynamic Programming:** Use a bottom-up approach to find subsets that sum up to the target value.
* **Backtracking:** Explore possible subsets, pruning branches that exceed the target value.

**Objective:**

The objective of this experiment is to solve the Sum of Subset Problem, where a subset of a given set SSS is identified such that the sum of the elements in the subset equals a given value.

**Algorithm**:

* + Start with an empty subset and a sum of 0.
  + Try to add each element of SSS to the current subset.
  + If the sum exceeds d, backtrack and remove the element.
  + If the sum equals d, record the subset.
  + Continue until all subsets are explored.

Code Implementation:

#include <iostream>

#include <cmath>

using namespace std;

int main() {

int N, target\_sum;

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

cin >> N;

int S[N];

cout << "Enter the elements: ";

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

cin >> S[i];

}

cout << "Enter the target sum: ";

cin >> target\_sum;

int total\_subsets = 1 << N;

int count = 0;

for (int mask = 0; mask < total\_subsets; mask++) {

int subset\_sum = 0;

bool found = false;

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

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

subset\_sum += S[j];

}

}

if (subset\_sum == target\_sum) {

found = true;

cout << "{ ";

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

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

cout << S[j] << " ";

}

}

cout << "}\n";

count++;

}

}

cout << "Total subsets found: " << count << endl;

return 0;

}  
Output :

Enter the number of elements: 6

Enter the elements: 5 10 12 13 15 18

Enter the target sum: 30

{ 5 12 13 }

{ 5 10 15 }

{ 12 18 }

Total subsets found: 3

**Lab Report 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.**

**Theory:**

In the 0/1 Knapsack Problem, we are given a set of items, each with a profit and a weight. We also have a knapsack with a maximum weight capacity. The goal is to determine the maximum profit we can achieve by selecting a subset of items such that their total weight does not exceed the capacity.

**Objective:**

The objective of this experiment is to solve the **0/1 Knapsack Problem** using the dynamic programming approach.

Algorithm:

 **Input**: Read the number of items nnn, their profits, weights, and the knapsack capacity.

 **Initialize DP Table**: Create a 2D array dp where dp[i][w] stores the maximum profit for the first iii items with capacity www.

 **Base Case**: Set dp[0][w] = 0 and dp[i][0] = 0 for all iii and www, as the profit is 0 when there are no items or capacity.

 **Fill DP Table**: Loop through each item and capacity:

* If the item fits in the knapsack (weight ≤\leq≤ current capacity), update dp[i][w] as the maximum of including or excluding the item.
* Otherwise, carry forward the value from the previous item (dp[i][w] = dp[i-1][w]).

 **Display DP Table**: Output the filled DP table to visualize the decision-making process.

 **Trace Items Included**: Starting from dp[n][capacity], trace back the items included by comparing dp[i][w] with dp[i-1][w].

 **Output Included Items**: Display the items selected and their profit and weight.

 **Output Maximum Profit**: The value in dp[n][capacity] is the maximum profit. Output this value.

Code Implementation:

#include <iostream>

using namespace std;

int main() {

int profits[100], weights[100];

int dp[101][101];

int n, capacity;

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

cin >> n;

cout << "Enter the profits of the items:\n";

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

cin >> profits[i];

}

cout << "Enter the weights of the items:\n";

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

cin >> weights[i];

}

cout << "Enter the capacity of the knapsack: ";

cin >> capacity;

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

for (int w = 0; w <= capacity; w++) {

if (i == 0 || w == 0) {

dp[i][w] = 0;

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

dp[i][w] = (profits[i - 1] + dp[i - 1][w - weights[i - 1]] > dp[i - 1][w]) ?

profits[i - 1] + dp[i - 1][w - weights[i - 1]] : dp[i - 1][w];

} else {

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

}

}

}

cout << "DP Table:\n";

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

for (int w = 0; w <= capacity; w++) {

cout << dp[i][w] << " ";

}

cout << endl;

}

int w = capacity;

cout << "\nItems included to achieve maximum profit:\n";

for (int i = n; i > 0; i--) {

if (dp[i][w] != dp[i - 1][w]) {

cout << "Item " << i << " (Profit: " << profits[i - 1] << ", Weight: " << weights[i - 1]

<< ")\n";

w = w - weights[i - 1];

}

}

cout << "Maximum profit in the knapsack: " << dp[n][capacity] << endl;

return 0;

}

Output :

Enter the number of items: 4

Enter the profits of the items:

15 25 13 23

Enter the weights of the items:

2 6 12 9

Enter the capacity of the knapsack: 20

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 to achieve 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

**Lab Report 10:** Write a program to solve the **Tower of Hanoi** problem for the **N** disk

**Objective**:  
To solve the Tower of Hanoi problem using recursion for NNN disks.

**Theory:**

The Tower of Hanoi is a classic puzzle consisting of three rods and a number of disks of different sizes that can slide onto any rod. Initially, all the disks are stacked in decreasing size on one rod. The puzzle requires moving all the disks from one rod to another, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be moved to the top of another rod if it is smaller than the disk already on that rod.
3. All disks start on the first rod and need to be moved to the last rod.

The problem can be solved recursively by breaking it into smaller sub-problems:

1. Move N−1N-1N−1 disks from the source rod to the auxiliary rod.
2. Move the NNN-th (largest) disk to the destination rod.
3. Move the N−1N-1N−1 disks from the auxiliary rod to the destination rod.

**Algorithm:**

1. **Input**: Number of disks NNN, source rod AAA, auxiliary rod BBB, and destination rod CCC.
2. **Base Case**: If N=1N = 1N=1, move the disk directly from source rod AAA to destination rod CCC.
3. **Recursive Case**: For N>1N > 1N>1:
   * Move N−1N-1N−1 disks from source rod AAA to auxiliary rod BBB.
   * Move the largest disk (disk NNN) from source rod AAA to destination rod CCC.
   * Move the N−1N-1N−1 disks from auxiliary rod BBB to destination rod CCC.
4. **Output**: Display each move made during the process.

Code Implementation:

#include <iostream>

using namespace std;

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

if (N == 1) {

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

return;

}

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

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

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

}

int main() {

int N;

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

cin >> N;

towerOfHanoi(N, 'A', 'B', 'C');

return 0;

}

Input:

Enter the number of disks: 3

Output:

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