Pabna University Of Science & Technology

Information & Communication Engineering

Course Title:Data Structure & Algorithm Sessional

Course Code:DSA-2202

LAB REPORT

Submitted By:

Name:Md.Abu-Bakkar Siddik Zilhaj

Roll:220617

Session :2021-2022

Submitted To:

Dr.Md.Anwar Hossain

Associate Professor

Dept. Of ICE,PUST

Date Of Issue:26-2-2025

Date of Submission:28-2-2025

Index

LAB REPORT

Course Code: ICE-2202
Course Title: Data Structure and Algorithm Sessional

Source Code:DSA-2202

Lab 1:

TITLE: *Bubble Sort Algorithm Implementation and Analysis*

OBJECTIVE: *To implement the Bubble Sort algorithm and analyze its working, efficiency, and output.*

THEORY

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the array while larger elements sink to the bottom with each pass.

Working Principle:

1. *Start at the first element and compare it with the next.*
2. *If the first element is greater than the second, swap them.*
3. *Move to the next pair and repeat the process.*
4. *Continue until the end of the array.*
5. *Repeat the process for the remaining elements until the list is completely sorted.*

Time Complexity Analysis:

- ***Best Case (Already Sorted):*** *$O(n)$*
- ***Average Case:*** *$O(n^2)$*
- ***Worst Case (Reverse Sorted):*** *$O(n^2)$*

Space Complexity:

- *$O(1)$ (since sorting is done in place, requiring no extra space)*

ALGORITHM

1. *Start from the first index of the array.*

2. *Compare the current element with the next element.*
3. *Swap them if the current element is greater than the next element.*
4. *Repeat steps 2-3 for the entire array, reducing the size of the unsorted portion after each full pass.*
5. *Continue the process until no swaps are required.*
6. *The array is now sorted.*

SOURCE CODE (C Program for Bubble Sort)

```
#include <stdio.h>

// Function to implement Bubble Sort
void bubbleSort(int arr[], int n) {
   int i, j, temp;
   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
```

```
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

INPUT & OUTPUT

Input:

```
Original array:
64 34 25 12 22 11 90
```

Sorting Process (Step by Step):

1st Pass: [34, 25, 12, 22, 11, 64, 90] 2nd Pass: [25, 12, 22, 11, 34, 64, 90] 3rd Pass: [12, 22, 11, 25, 34, 64, 90] 4th Pass: [12, 11, 22, 25, 34, 64, 90] 5th Pass: [11, 12, 22, 25, 34, 64, 90]

Output:

Sorted array:

11 12 22 25 34 64 90

Lab 2: ***LAB REPORT ON LINEAR SEARCH ALGORITHM***

TITLE: *Linear Search Algorithm Implementation and Analysis*

OBJECTIVE: *To implement the Linear Search algorithm and analyze its working, efficiency, and output.*

THEORY

Linear Search is a simple searching algorithm that checks each element of the list sequentially until the desired element is found or the list ends. It is widely used for small datasets or unsorted lists due to its simplicity.

Working Principle:

1. *Start from the first element of the array.*
2. *Compare the current element with the target element.*
3. *If a match is found, return the index of the element.*
4. *If no match is found, continue searching until the end of the array.*
5. *If the element is not present, return -1 or an appropriate message.*

Time Complexity Analysis:

- ***Best Case (Element Found at Start):*** *O(1)*
- ***Average Case:*** *O(n)*

- ***Worst Case (Element Not Found or at End):*** *O(n)*

Space Complexity:

- *O(1) (since the search is conducted in place without extra storage)*

ALGORITHM

1. *Start at the first index of the array.*
2. *Compare the current element with the target element.*
3. *If the element is found, return its index.*
4. *If not found, move to the next element.*
5. *Repeat the process until the end of the array.*
6. *If the element is not found, return an appropriate message.*

SOURCE CODE (C Program for Linear Search)

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

```
// Function to implement Linear Search
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i; // Return index if element found
        }
    }
    return -1; // Return -1 if element not found
}

// Main function
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 30;

    int result = linearSearch(arr, n, target);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

INPUT & OUTPUT

Input:

Array elements:

10 20 30 40 50

Target element:

30

Searching Process (Step by Step):

1st Comparison: 10 != 30 2nd Comparison: 20 != 30 3rd Comparison: 30 == 30 (Match Found)

Output:

Element found at index 2

Lab 3: ***LAB REPORT ON MERGE SORT ALGORITHM***

TITLE: *Merge Sort Algorithm Implementation and Analysis*

OBJECTIVE: *To implement the Merge Sort algorithm and analyze its working, efficiency, and output.*

THEORY

Merge Sort is a divide-and-conquer sorting algorithm that recursively divides the array into smaller subarrays, sorts them, and then merges the sorted subarrays to form a fully sorted array. It is one of the most efficient sorting techniques, particularly useful for large datasets.

Working Principle:

1. *Divide the array into two halves recursively until each subarray contains a single element.*
2. *Merge the subarrays back together in a sorted order.*
3. *Continue merging until the entire array is sorted.*

Time Complexity Analysis:

- ***Best Case:*** *O(n log n)*
- ***Average Case:*** *O(n log n)*
- ***Worst Case:*** *O(n log n)*

Space Complexity:

- *O(n) (due to the auxiliary space used for merging)*

ALGORITHM

1. *If the array has one or zero elements, it is already sorted.*
2. *Divide the array into two halves.*
3. *Recursively sort both halves using Merge Sort.*

4. Merge the two sorted halves into a single sorted array.
5. Repeat the process until the entire array is sorted.

SOURCE CODE (C Program for Merge Sort)

```
#include <stdio.h>

// Function to merge two halves of an array
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
```

```
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++;
}

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to implement Merge Sort
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);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
```

}

INPUT & OUTPUT

Input:

Original array:

38 27 43 3 9 82 10

Sorting Process (Step by Step):

1. *Split into [38, 27, 43, 3] and [9, 82, 10]*
2. *Further split into [38, 27], [43, 3], [9, 82], [10]*
3. *Merge [38, 27] → [27, 38], [43, 3] → [3, 43], [9, 82] remains unchanged.*
4. *Merge [27, 38] with [3, 43] → [3, 27, 38, 43]*
5. *Merge [9, 82] with [10] → [9, 10, 82]*
6. *Merge [3, 27, 38, 43] with [9, 10, 82] → [3, 9, 10, 27, 38, 43, 82]*

Output:

Sorted array:

3 9 10 27 38 43 82

Lab 4: ***LAB REPORT ON BINARY SEARCH ALGORITHM***

TITLE: *Binary Search Algorithm Implementation and Analysis*

OBJECTIVE: *To implement the Binary Search algorithm and analyze its working, efficiency, and output.*

THEORY

Binary Search is an efficient searching algorithm used to find an element in a sorted array. It follows the divide-and-conquer approach by repeatedly dividing the search interval in half. The search begins by comparing the target element with the middle element of the array. If a match is found, the search terminates; otherwise, the process continues *in the left or right half, depending on whether the target is smaller or greater than the middle element.*

Working Principle:

1. *The array must be sorted before applying Binary Search.*
2. *Find the middle element of the array.*
3. *If the middle element matches the target, return the index.*
4. *If the middle element is greater than the target, search the left half.*
5. *If the middle element is smaller than the target, search the right half.*
6. *Repeat the process until the element is found or the search space is empty.*

Time Complexity Analysis:

- ***Best Case:*** *O(1) (When the element is found in the first comparison)*
- ***Average Case:*** *O(log n)*
- ***Worst Case:*** *O(log n)*

Space Complexity:

- *O(1) for iterative approach*
- *O(log n) for recursive approach (due to recursion stack)*

ALGORITHM

1. *Start with low = 0 and high = n - 1.*
2. *Find the middle index using mid = low + (high - low) / 2.*
3. *If arr[mid] is equal to the target, return mid.*
4. *If arr[mid] is greater than the target, update high = mid - 1.*
5. *If arr[mid] is less than the target, update low = mid + 1.*
6. *Repeat steps 2-5 until low > high.*
7. *If the element is not found, return -1.*

SOURCE CODE (C Program for Binary Search)

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

```
// Function to implement Binary Search
int binarySearch(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid; // Target found
        else if (arr[mid] < target)
            low = mid + 1; // Search right half
        else
            high = mid - 1; // Search left half
    }
    return -1; // Target not found
}

// Main function
int main() {
    int arr[] = {10, 20, 30, 40, 50, 60, 70};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 40;

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

    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

INPUT & OUTPUT

Input:

Array elements:

10 20 30 40 50 60 70

Target element:

40

Searching Process (Step by Step):

1. *mid = 3 (Element at index 3 is 40, which matches the target).*
2. *Element found at index 3.*

Output:

Element found at index 3

Lab 5:

LAB REPORT ON PATTERN MATCHING ALGORITHM

***TITLE:** Pattern Matching Algorithm Implementation and Analysis*

OBJECTIVE: *To implement a Pattern Matching algorithm and analyze its working, efficiency, and output.*

THEORY

Pattern Matching is a fundamental problem in computer science used to find occurrences of a specific pattern within a given text. It is widely applied in search engines, text processing, and bioinformatics.

Working Principle:

1. *Given a text T of length n and a pattern P of length m, the goal is to find all occurrences of P in T.*
2. *The algorithm slides the pattern over the text, comparing characters at each step.*
3. *If a match is found, the starting index of the match is recorded.*
4. *Efficient algorithms such as the Naïve Approach, Knuth-Morris-Pratt (KMP), and Boyer-Moore optimize the search process.*

Time Complexity Analysis:

- ***Naïve Approach:*** *O(n * m)*
- ***Knuth-Morris-Pratt (KMP):*** *O(n + m)*
- ***Boyer-Moore:*** *O(n / m)*

Space Complexity:

- *O(1) for Naïve Approach*
- *O(m) for KMP (due to preprocessing table)*

ALGORITHM (Knuth-Morris-Pratt - KMP)

1. *Preprocess the pattern to create a longest prefix suffix (LPS) array.*
2. *Initialize indices i = 0 (text) and j = 0 (pattern).*
3. *Compare P[j] with T[i].*
4. *If they match, increment both indices; if j reaches m, record the match.*
5. *If they do not match, use the LPS array to shift the pattern efficiently.*
6. *Repeat until i reaches n.*

SOURCE CODE (C Program for Pattern Matching using KMP Algorithm)

```
#include <stdio.h>

#include <string.h>


void computeLPSArray(char* pattern, int m, int*
lps) {

    int len = 0;

    lps[0] = 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];

            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

void KMPSearch(char* text, char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int lps[m];
    computeLPSArray(pattern, m, lps);

    int i = 0, j = 0;
    while (i < n) {
        if (pattern[j] == text[i]) {
            i++, j++;
        }
        if (j == m) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    char text[] = "ABABDABACDABABCABAB";
    char pattern[] = "ABABCABAB";
    KMPSearch(text, pattern);
    return 0;
```

}

INPUT & OUTPUT

Input:

Text:

ABABDABACDABABCABAB

Pattern:

ABABCABAB

Matching Process (Step by Step):

1. *Compute LPS array for pattern.*
2. *Compare pattern with text starting at each position.*
3. *Shift pattern using LPS array when mismatches occur.*
4. *Match found at index 10.*

Output:

Pattern found at index 10

Lab 6:

LAB REPORT ON QUEUE DATA STRUCTURE IMPLEMENTATION

TITLE: *Queue Data Structure Implementation and Analysis*

OBJECTIVE: *To implement a Queue data structure and analyze its working, efficiency, and output.*

THEORY

A Queue is a linear data structure that follows the First In First Out (FIFO) principle. It means that the element inserted first will be the first one to be removed. Queues are widely used in computer science applications such as scheduling processes in operating systems, handling requests in web servers, and managing tasks in printer spooling.

Types of Queues:

1. ***Simple Queue*** *- Elements are added at the rear and removed from the front.*

2. ***Circular Queue*** *- The last element connects back to the first element to utilize space efficiently.*
3. ***Priority Queue*** *- Elements are dequeued based on priority rather than order of insertion.*
4. ***Double-Ended Queue (Deque)*** *- Elements can be inserted or removed from both ends.*

Operations on Queue:

1. ***Enqueue:*** *Adds an element to the rear of the queue.*
2. ***Dequeue:*** *Removes an element from the front of the queue.*
3. ***Peek:*** *Retrieves the front element without removing it.*
4. ***IsEmpty:*** *Checks if the queue is empty.*
5. ***IsFull:*** *Checks if the queue is full.*

Time Complexity Analysis:

- ***Enqueue:*** *O(1)*
- ***Dequeue:*** *O(1)*
- ***Peek:*** *O(1)*

- ***IsEmpty/IsFull:*** *O(1)*

Space Complexity:

- *O(n) (for storing n elements)*

ALGORITHM

1. ***Initialize*** *an array-based queue with a fixed size.*
2. ***Define*** *front and rear pointers.*
3. ***Enqueue Operation:***
   - *Check if the queue is full.*
   - *Insert the element at the rear.*
   - *Update the rear pointer.*
4. ***Dequeue Operation:***
   - *Check if the queue is empty.*
   - *Remove the element from the front.*
   - *Update the front pointer.*
5. ***Peek Operation:***
   - *Return the front element without dequeuing.*
6. ***Check Empty/Full Condition:***

- *If front == -1, queue is empty.*
- *If rear == size - 1, queue is full.*

SOURCE CODE (C Program for Queue Implementation)

```
#include <stdio.h>
#define SIZE 5

// Queue structure
struct Queue {
    int items[SIZE];
    int front, rear;
};

// Initialize queue
void initializeQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
```

```
}

// Check if queue is empty
int isEmpty(struct Queue* q) {
    return q->front == -1;
}

// Check if queue is full
int isFull(struct Queue* q) {
    return q->rear == SIZE - 1;
}

// Enqueue operation
void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
    printf("Inserted %d\n", value);
}

// Dequeue operation
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }
    int item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    return item;
```

```
}

// Peek operation
int peek(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }
    return q->items[q->front];
}

// Main function
int main() {
    struct Queue q;
    initializeQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);

    printf("Front element: %d\n", peek(&q));

    printf("Dequeued: %d\n", dequeue(&q));
    printf("Dequeued: %d\n", dequeue(&q));

    printf("Front element after dequeues: %d\n",
peek(&q));

    return 0;
}
```

INPUT & OUTPUT

Input:

Operations performed in sequence:

1. *Enqueue 10*
2. *Enqueue 20*
3. *Enqueue 30*
4. *Enqueue 40*
5. *Enqueue 50*
6. *Peek*
7. *Dequeue*
8. *Dequeue*
9. *Peek*

Output:

```
Inserted 10
Inserted 20
Inserted 30
Inserted 40
Inserted 50
Front element: 10
Dequeued: 10
Dequeued: 20
```

Front element after dequeues: 30

Lab 7:

LAB REPORT ON N-QUEENS PROBLEM USING BACKTRACKING

TITLE: *Solving the N-Queens Problem Using Backtracking*

OBJECTIVE: *To implement the N-Queens problem using the Backtracking algorithm and analyze its working, efficiency, and output.*

THEORY

The N-Queens problem is a classic combinatorial problem that involves placing N queens on an N × N chessboard such that no two queens threaten each other. This means that:

- *No two queens should be in the same row.*

- *No two queens should be in the same column.*
- *No two queens should be on the same diagonal.*

The problem can be efficiently solved using the Backtracking algorithm, which explores all possible placements of queens on the board and backtracks whenever a conflict is found.

Concept of Backtracking:

Backtracking is an algorithmic technique for solving recursive problems by trying possible solutions and eliminating those that fail to satisfy the constraints.

1. *Start placing queens one by one in different rows.*
2. *Check if placing a queen in a particular position is safe.*
3. *If safe, place the queen and proceed to place the next queen.*
4. *If not safe, backtrack and try the next available position.*
5. *Repeat the process until all queens are placed successfully.*

Time Complexity Analysis:

- *The worst-case time complexity is **O(N!)** since all possible arrangements of queens are considered.*

Space Complexity:

- ***O(N^2)*** *for the board representation.*
- ***O(N)*** *for storing column and diagonal constraints.*

ALGORITHM

1. *Create an N × N chessboard initialized to zero.*
2. *Define a recursive function solveNQueens(row, board).*
3. *If row == N, print the board (solution found).*
4. *For each column in the current row:*
   - *Check if placing a queen at board[row][col] is safe.*
   - *If safe, place the queen (board[row][col] = 1).*
   - *Recursively call solveNQueens for the next row.*

    - *If a solution is not found, backtrack (board[row][col] = 0).*
5. *If all rows are processed successfully, return True; otherwise, return False.*

SOURCE CODE (C Program for N-Queens Using Backtracking)

```
#include <stdio.h>
#define N 8

void printSolution(int board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      printf("%d ", board[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
```

```
int isSafe(int board[N][N], int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i][col])
            return 0;
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
{
        if (board[i][j])
            return 0;
    }
    for (int i = row, j = col; i >= 0 && j < N; i--, j++)
{
        if (board[i][j])
            return 0;
    }
    return 1;
}
```

```
int solveNQueens(int board[N][N], int row) {
    if (row >= N) {
        printSolution(board);
        return 1;
    }
    for (int i = 0; i < N; i++) {
        if (isSafe(board, row, i)) {
            board[row][i] = 1;
            solveNQueens(board, row + 1);
            board[row][i] = 0;
        }
    }
    return 0;
}

int main() {
    int board[N][N] = {0};
    solveNQueens(board, 0);
    return 0;
}
```

INPUT & OUTPUT

Input:

The program takes N = 8 as input (can be modified for different values of N).

Output:

For N = 4, one possible solution:

```
0  1  0  0
0  0  0  1
1  0  0  0
0  0  1  0
```

For N = 8, the program prints all possible valid arrangements of 8 queens.

Lab 8:

LAB REPORT ON SUM OF SUBSETS PROBLEM USING BACKTRACKING

TITLE: *Solving the Sum of Subsets Problem Using Backtracking*

OBJECTIVE: *To implement the Sum of Subsets problem using the Backtracking algorithm and analyze its working, efficiency, and output.*

THEORY

The Sum of Subsets problem is a combinatorial problem where given a set of positive integers and a target sum d, we aim to find all subsets whose sum is exactly equal to d. The problem can be effectively solved using the backtracking technique, which systematically explores all possible subsets while eliminating those that cannot lead to a valid solution early in the search process.

Concept of Backtracking:

Backtracking is a problem-solving technique that involves exploring all possible options and discarding those that fail to satisfy the given constraints. It works by making a series of choices and undoing those choices when they lead to a dead end.

Mathematical Formulation:

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

Time Complexity Analysis:

- *The worst-case time complexity is **O(2^N)**, as all subsets are explored in the worst case.*

Space Complexity:

- ***O(N)**, required for recursion stack and subset storage.*

ALGORITHM

1. *Sort the given set in non-decreasing order.*
2. *Define a recursive function sumOfSubsets(current_sum, index, remaining_sum, subset).*

3. *If current_sum == d, print the subset (valid solution found).*
4. *If current_sum > d or index == n, return (backtrack).*
5. *Include the current element in the subset and make a recursive call.*
6. *Exclude the current element and make another recursive call.*
7. *Repeat until all valid subsets are found.*

SOURCE CODE (C Program for Sum of Subsets Using Backtracking)

```
#include <stdio.h>
#define N 6


int S[N] = {5, 10, 12, 13, 15, 18};
int d = 30;


void printSubset(int subset[], int size) {
    printf("{ ");
    for (int i = 0; i < size; i++) {
        printf("%d ", subset[i]);
    }
    printf("}\n");
}

void sumOfSubsets(int current_sum, int index, int
remaining_sum, int subset[], int subset_size) {
    if (current_sum == d) {
        printSubset(subset, subset_size);
        return;
    }
    if (index == N || current_sum > d) {
        return;
    }

    if (current_sum + S[index] <= d) {
        subset[subset_size] = S[index];
        sumOfSubsets(current_sum + S[index], index + 1, remaining_sum - S[index], subset, subset_size + 1);
    }

    sumOfSubsets(current_sum, index + 1, remaining_sum - S[index], subset, subset_size);
}

int main() {
    int subset[N];
    int total_sum = 0;
    for (int i = 0; i < N; i++) {
        total_sum += S[i];
    }

    sumOfSubsets(0, 0, total_sum, subset, 0);
    return 0;
```

}

INPUT & OUTPUT

Input:

Set S = {5,10,12,13,15,18}, target sum d = 30

Output:

{ 5 10 15 }

{ 5 12 13 }

{ 12 18 }

Lab 9:

LAB REPORT ON 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

TITLE: *Solving the 0/1 Knapsack Problem Using Dynamic Programming*

OBJECTIVE: *To implement the 0/1 Knapsack problem using the Dynamic Programming approach and analyze its efficiency and output.*

THEORY

The 0/1 Knapsack problem is a fundamental problem in combinatorial optimization. Given a set of items, each with a weight and a profit, the goal is to determine the maximum profit that can be obtained by selecting a subset of these items, ensuring that their total weight does not exceed the given knapsack capacity.

Problem Definition:

Given:

- *n items, each with a weight w[i] and profit p[i].*
- *A knapsack with maximum weight capacity W.*
- *Objective: Maximize the total profit while ensuring that the sum of selected item weights does not exceed W.*

Dynamic Programming Approach:

Dynamic programming (DP) is used to solve this problem optimally by building a table that stores the maximum profit for subproblems of different capacities and item subsets.

Mathematical Formulation:

Let dp[i][j] represent the maximum profit achievable using the first i items and a knapsack capacity j. The recurrence relation is:

dp[i][j]=max(p[i]+dp[i−1][j−w[i]],dp[i−1][j])dp[i][j] = \max (p[i] + dp[i-1][j - w[i]], dp[i-1][j])

where:

- *p[i] + dp[i-1][j - w[i]] represents the case where the i-th item is included.*
- *dp[i-1][j] represents the case where the i-th item is excluded.*

Time Complexity Analysis:

- *The time complexity is **O(n * W)**, where n is the number of items and W is the knapsack capacity.*
- *The space complexity is **O(n * W)** due to the storage of the DP table.*

ALGORITHM

1. *Create a DP table dp[n+1][W+1] initialized to 0.*
2. *Iterate over items and capacities to fill the table:*
   - *If an item's weight is less than the current capacity, update the table using the recurrence relation.*
   - *Otherwise, retain the previous value.*
3. *The bottom-right cell of the table contains the maximum profit.*
4. *Backtrack to find the selected items.*

SOURCE CODE (C Program for 0/1 Knapsack Using Dynamic Programming)

```
#include <stdio.h>


#define MAX_ITEMS 100
#define MAX_WEIGHT 100
```

```
int max(int a, int b) { return (a > b) ? a : b; }

void knapsack(int profits[], int weights[], int n, int W)
{
    int dp[n+1][W+1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else 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];
        }
    }
    printf("Maximum Profit: %d\n", dp[n][W]);

    printf("Selected items: ");
    int w = W;
    for (int i = n; i > 0 && dp[i][w] != 0; i--) {
        if (dp[i][w] != dp[i-1][w]) {
            printf("Item %d ", i);
            w -= weights[i-1];
        }
    }
    printf("\n");
}

int main() {
    int profits[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(profits) / sizeof(profits[0]);
    knapsack(profits, weights, n, W);
    return 0;
}
```

INPUT & OUTPUT

Input:

Profits: {60, 100, 120} Weights: {10, 20, 30} Knapsack Capacity: 50

Output:

```
Maximum Profit: 220
Selected items: Item 3 Item 2
```

Lab 10:

LAB REPORT ON TOWER OF HANOI USING RECURSION

TITLE: *Solving the Tower of Hanoi Problem Using Recursion*

OBJECTIVE: *To implement the Tower of Hanoi problem using a recursive approach and analyze its efficiency and output.*

THEORY

The Tower of Hanoi is a classic problem in the field of recursion and algorithm design. It consists of three rods (towers) and N disks of different sizes stacked in decreasing order on one of the rods. The objective is to move all the disks from the source rod to the destination rod using an auxiliary rod, following these rules:

1. *Only one disk can be moved at a time.*
2. *A larger disk cannot be placed on top of a smaller disk.*
3. *Only the topmost disk of a rod can be moved.*

Mathematical Formulation:

For N disks, the number of moves required is given by the recurrence relation:

T(N)=2T(N−1)+1T(N) = 2T(N-1) + 1

Expanding the recurrence, the total number of moves required is:

T(N)=2N−1T(N) = 2^N - 1

Time Complexity Analysis:

- *The time complexity of the recursive solution is **O(2^N)**.*
- *The space complexity is **O(N)** due to the recursive function call stack.*

ALGORITHM

1. *Define a recursive function towerOfHanoi(N, source, auxiliary, destination).*
2. *Base Case: If N == 1, move the disk from source to destination and return.*
3. *Recursively move N-1 disks from source to auxiliary using destination as an intermediary.*
4. *Move the Nth disk from source to destination.*
5. *Recursively move N-1 disks from auxiliary to destination using source as an intermediary.*

6. *Repeat until all disks are moved to the destination rod.*

SOURCE CODE (C Program for Tower of Hanoi Using Recursion)

```
#include <stdio.h>


void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }


    towerOfHanoi(n - 1, source, destination, auxiliary);
    printf("Move disk %d from %c to %c\n", n, source, destination);
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int N;
    printf("Enter the number of disks: ");
    scanf("%d", &N);
    printf("\nSteps to solve Tower of Hanoi for %d disks:\n", N);
    towerOfHanoi(N, 'A', 'B', 'C');
    return 0;
}
```

INPUT & OUTPUT

Input:

```
Enter the number of disks: 3
```

Output:

```
Steps to solve Tower of Hanoi for 3 disks:
```

```
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
```