

Pabna University of Science and

TECHNOlOGY

**INFORMATION AND COMMUNICATION ENGINEERING**

Course Name: Data Structure and Algorithm Sessional

Course Code: ICE-2202

Submitted To,

Professor Md.Anwar hossain

DEpARTwENT OF ICE, PUST

Submitted By, MD.MURSAlIN LEMON Roll: 220606 SESSION:2021-22

2nd Year 2nd Semester

DEpARTWENT OF ICE, PUST

|  |  |
| --- | --- |
| 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. |

# Problem 1: Bubble Sort Algorithm Illustration:

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 pass through the list is repeated until the list is sorted.

Example: Consider the array [5, 3, 8, 4, 6].

* First Pass:
  + Compare 5 and 3: swap to get [3, 5, 8, 4, 6].
  + Compare 5 and 8: no swap.
  + Compare 8 and 4: swap to get [3, 5, 4, 8, 6].
  + Compare 8 and 6: swap to get [3, 5, 4, 6, 8].
* Second Pass:
  + Compare 3 and 5: no swap.
  + Compare 5 and 4: swap to get [3, 4, 5, 6, 8].
  + Compare 5 and 6: no swap.
* The array is now sorted as [3, 4, 5, 6, 8].

# Algorithm:

1. Start from the first element, compare it with the next element.
2. If the current element is greater than the next element, swap them.
3. Move to the next pair of elements and repeat the process.
4. Continue this process until no more swaps are needed.

# Source Code:

public class BubbleSort {

public static void bubbleSort(int[] arr) { int n = arr.length;

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

for (int j = 0; j < n-i-1; j++) if (arr[j] > arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j+1]; arr[j+1] = temp;

}

}

public static void main(String[] args) { int[] arr = {64, 34, 25, 12, 22, 11, 90};

bubbleSort(arr); System.out.println("Sorted array:"); for (int i : arr) {

System.out.print(i + " ");

}

}

}

# Sample Input and Output:

* Input: {64, 34, 25, 12, 22, 11, 90}
* Output: Sorted array: 11 12 22 25 34 64 90

# Problem 2: Linear Search Algorithm Illustration:

Linear search is a method for finding a particular value in a list. It checks each element of the

list one by one from the start until the desired element is found. This algorithm is straightforward but can be inefficient for large lists.

Example: Consider the array [10, 20, 30, 40, 50] and we want to find 30.

* + Start at the first element: 10 (not a match).
  + Move to the second element: 20 (not a match).
  + Move to the third element: 30 (match found at index 2).

# Algorithm:

1. Start from the first element of the array.
2. Compare the current element with the target value.
3. If the current element matches the target value, return the index.
4. If the end of the array is reached without finding the target, return -1.

# Source Code:

public class LinearSearch {

public static int linearSearch(int[] arr, int target) { for (int i = 0; i < arr.length; i++) {

if (arr[i] == target) { return i;

}

}

return -1;

}

public static void main(String[] args) { int[] arr = {10, 20, 30, 40, 50};

int target = 30;

int result = linearSearch(arr, target); if (result == -1) {

System.out.println("Element not found");

} else {

System.out.println("Element found at index: " + result);

}

}

}

# Sample Input and Output:

* Input: {10, 20, 30, 40, 50}, Target: 30
* Output: Element found at index: 2

# Problem 3: Merge Sort Algorithm

**Illustration:**

Merge sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the two sorted halves. This algorithm is efficient and has a time complexity of O(n log n).

Example: Consider the array [38, 27, 43, 3, 9, 82, 10].

* + Divide:
    - Split into [38, 27, 43, 3] and [9, 82, 10].
  + Conquer:
    - Recursively sort each half: [3, 27, 38, 43] and [9, 10, 82].
  + Combine:
    - Merge the two halves to get [3, 9, 10, 27, 38, 43, 82].

# Algorithm:

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

# Source Code:

public class MergeSort {

public static void mergeSort(int[] arr, int left, int right) { if (left < right) {

int mid = (left + right) / 2; mergeSort(arr, left, mid); mergeSort(arr, mid + 1, right); merge(arr, left, mid, right);

}

}

public static void merge(int[] arr, int left, int mid, int right) { int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1]; int[] R = new int[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; int 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++;

}

}

public static void main(String[] args) { int[] arr = {38, 27, 43, 3, 9, 82, 10};

mergeSort(arr, 0, arr.length - 1); System.out.println("Sorted array:"); for (int i : arr) {

System.out.print(i + " ");

}

}

}

# Sample Input and Output:

* Input: {38, 27, 43, 3, 9, 82, 10}
* Output: Sorted array: 3 9 10 27 38 43 82

# Problem 4: Binary Search Algorithm Illustration:

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

Example: Consider the sorted array [1, 3, 5, 7, 9, 11] and we want to find 7.

* + First Step: Middle element is 5 (index 2). Since 7 > 5, search the right half [7, 9, 11].
  + Second Step: Middle element is 9 (index 4). Since 7 < 9, search the left half [7].
  + Third Step: Middle element is 7 (index 3). Match found.

# Algorithm:

1. Compare the target value to the middle element of the array.
2. If the target value is equal to the middle element, return the index.
3. If the target value is less than the middle element, repeat the search in the left half.
4. If the target value is greater than the middle element, repeat the search in the right half.
5. If the search ends with the remaining half being empty, the target is not in the array.

# Source Code:

public class BinarySearch {

public static int binarySearch(int[] arr, int target) { int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2; if (arr[mid] == target) {

return mid;

}

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

} else {

right = mid - 1;

}

}

return -1;

}

public static void main(String[] args) { int[] arr = {2, 3, 4, 10, 40};

int target = 10;

int result = binarySearch(arr, target); if (result == -1) {

System.out.println("Element not found");

} else {

System.out.println("Element found at index: " + result);

}

}

}

# Sample Input and Output:

* Input: {2, 3, 4, 10, 40}, Target: 10
* Output: Element found at index: 3

# Problem 5: Pattern Matching Algorithm Illustration:

Pattern matching involves finding a substring (pattern) within a larger string (text). The algorithm checks for the presence of the pattern at each position in the text. This is a fundamental operation in text processing.

Example: Consider the text "ABABDABACDABABCABAB" and the pattern "ABABC".

* + Start at the first character: "ABABD" (no match).
  + Move to the next character: "BABDA" (no match).
  + Continue until the substring "ABABC" matches the pattern at position 10.

# Algorithm:

1. Iterate through the text.
2. For each position in the text, check if the pattern matches starting at that position.
3. If a match is found, return the starting index.
4. If no match is found after checking all positions, return -1.

# Source Code:

public class PatternMatching {

public static int findPattern(String text, String pattern) { int n = text.length();

int m = pattern.length();

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

for (j = 0; j < m; j++) {

if (text.charAt(i + j) != pattern.charAt(j)) { break;

}

}

if (j == m) { return i;

}

}

return -1;

}

public static void main(String[] args) {

String text = "ABABDABACDABABCABAB";

String pattern = "ABABCABAB";

int result = findPattern(text, pattern); if (result == -1) {

System.out.println("Pattern not found");

} else {

System.out.println("Pattern found at index: " + result);

}

}

}

# Sample Input and Output:

* Input: Text: "ABABDABACDABABCABAB", Pattern: "ABABCABAB"
* Output: Pattern found at index: 10

# Problem 6: Queue Data Structure Illustration:

A queue is a linear data structure that follows the First In First Out (FIFO) principle. Elements are added at the rear (enqueue) and removed from the front (dequeue). This structure is commonly used in scenarios like task scheduling, buffering, and breadth-first search.

Example: Start with an empty queue.

* + Enqueue 1: Queue is [1].
  + Enqueue 2: Queue is [1, 2].
  + Enqueue 3: Queue is [1, 2, 3].
  + Dequeue: Remove 1, Queue is [2, 3].
  + Dequeue: Remove 2, Queue is [3].
  + Dequeue: Remove 3, Queue is empty.

# Algorithm:

1. Initialize an array to represent the queue and pointers for the front and rear.
2. Implement the enqueue operation to add elements at the rear.
3. Implement the dequeue operation to remove elements from the front.
4. Handle edge cases like queue overflow (full queue) and underflow (empty queue).

**Source** **Code:** public class Queue { private int[] arr;

private int front;

private int rear; private int capacity; private int count;

public Queue(int size) { arr = new int[size]; capacity = size; front = 0;

rear = -1;

count = 0;

}

public void enqueue(int item) { if (isFull()) {

System.out.println("Queue is full"); return;

}

rear = (rear + 1) % capacity; arr[rear] = item;

count++;

}

public int dequeue() { if (isEmpty()) {

System.out.println("Queue is empty"); return -1;

}

int item = arr[front];

front = (front + 1) % capacity; count--;

return item;

}

public boolean isFull() { return count == capacity;

}

public boolean isEmpty() { return count == 0;

}

public static void main(String[] args) { Queue queue = new Queue(5); queue.enqueue(10); queue.enqueue(20);

queue.enqueue(30);

System.out.println("Dequeued: " + queue.dequeue()); System.out.println("Dequeued: " + queue.dequeue());

}

}

# Sample Input and Output:

* Input: Enqueue 10, 20, 30
* Output: Dequeued: 10

Dequeued: 20

# Problem 7: N-Queens Problem Illustration:

The N-Queens problem involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. The problem is solved using backtracking.

Example: For N=4, one possible solution is:

* Place a queen at (1,2).
* Place a queen at (2,4).
* Place a queen at (3,1).
* Place a queen at (4,3).
* No two queens are in the same row, column, or diagonal.

# Algorithm:

1. Start in the leftmost column.
2. If all queens are placed, return true.
3. Try all rows in the current column.
4. If the queen can be placed safely, mark this [row, column] as part of the solution.
5. Recursively check if placing the queen leads to a solution.
6. If placing the queen doesn't lead to a solution, backtrack and try the next row.
7. If all rows have been tried and nothing worked, return false.

# Source Code:

public class NQueens { private int N;

private int[] board; // board[row] stores the column where the queen is placed

public NQueens(int N) { this.N = N;

board = new int[N]; // Array stores queen positions

}

public boolean solve(int row) {

if (row == N) { // All queens placed successfully printBoard();

return true;

}

for (int col = 0; col < N; col++) { if (isSafe(row, col)) {

board[row] = col; // Place queen

if (solve(row + 1)) { // Move to next row return true;

}

// Backtrack: If placing queen here didn't work, try next column

}

}

return false; // No valid position found, backtrack

}

private boolean isSafe(int row, int col) { for (int i = 0; i < row; i++) {

if (board[i] == col || Math.abs(board[i] - col) == Math.abs(i - row)) { return false; // Column or diagonal conflict

}

}

return true;

}

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

for (int j = 0; j < N; j++) { System.out.print(board[i] == j ? "Q " : ". ");

}

System.out.println();

}

System.out.println();

}

public static void main(String[] args) { NQueens nQueens = new NQueens(8); if (!nQueens.solve(0)) {

System.out.println("No solution found.");

}

}

}

# Sample Input and Output:

* Input: N = 8
* Output: A valid 8-Queens solution on an 8x8 board.

# Problem 8: Sum of Subset Problem Illustration:

The Sum of Subset Problem involves finding a subset of a given set of integers *S* that adds up to a specified sum *d*. This problem is solved using backtracking, which systematically explores all possible subsets to find the one that meets the condition.

Example: Given:

* Set *S*={5,10,12,13,15,18}.
* Target sum *d*=30.

Step-by-Step Backtracking:

1. Start with an empty subset and sum 0.
2. Include 5: Subset {5}, sum 5.
3. Include 10: Subset {5,10}, sum 15.
4. Include 15: Subset {5,10,15}, sum 30.

Solution Found: The subset {5,10,15} sums up to 30.

# Algorithm:

1. Use backtracking to explore all possible subsets.
2. For each element, decide whether to include it in the current subset.
3. If including the current element does not exceed the target sum, proceed recursively.
4. If a subset with the target sum is found, return true.

# Source Code:

import java.util.ArrayList; import java.util.List;

public class SubsetSum {

static boolean isSubsetSum(int[] set, int n, int sum, List<Integer> subset) {

// Base Case: If sum is 0, print the subset and return true if (sum == 0) {

System.out.println("Subset found: " + subset); return true;

}

// If no more elements left and sum is not 0, return false if (n == 0) return false;

// If the last element is greater than sum, ignore it if (set[n - 1] > sum)

return isSubsetSum(set, n - 1, sum, subset);

// Include the last element in the subset subset.add(set[n - 1]);

if (isSubsetSum(set, n - 1, sum - set[n - 1], subset))

return true; // Stop recursion once a valid subset is found subset.remove(subset.size() - 1); // Backtrack

// Exclude the last element and check return isSubsetSum(set, n - 1, sum, subset);

}

public static void main(String[] args) { int[] set = {5, 10, 12, 13, 15, 18};

int sum = 30;

List<Integer> subset = new ArrayList<>();

if (!isSubsetSum(set, set.length, sum, subset))

System.out.println("No subset found");

}

}

# Sample Input and Output:

* Input: Set: {5, 10, 12, 13, 15, 18}, Sum: 30
* Output: Found a subset with given sum

# Problem 9: 0/1 Knapsack Problem Illustration:

The 0/1 Knapsack Problem involves selecting items with given weights and values to maximize the total value without exceeding the knapsack's capacity. This problem is solved using dynamic programming, which builds a table to store intermediate results and avoid redundant calculations.

Example: Given:

* + Profits *P*=(15,25,13,23).
  + Weights *W*=(2,6,12,9).
  + Capacity *C*=20.

Step-by-Step Dynamic Programming:

* Create a table where rows represent items and columns represent capacities from 0 to 20.
* Fill the table by deciding whether to include or exclude each item at each capacity.
* For each cell, choose the maximum value between:
* Including the current item (if its weight ≤ current capacity).
* Excluding the current item. Solution Found:
  + Select items with profits 15 (weight 2) and 25 (weight 6).
  + Total profit: 15+25=40.
  + Total weight: 2+6=8 (within capacity).

# Algorithm:

1. Use dynamic programming to build a table where each entry represents the maximum value achievable with a given weight limit.
2. Iterate through each item and update the table based on whether including the item increases the total value.

# Source Code:

public class Knapsack {

public static int knapsack(int[] profits, int[] weights, int capacity) { int n = profits.length;

int[][] dp = new int[n + 1][capacity + 1];

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

for (int w = 0; w <= capacity; w++) { if (weights[i - 1] <= w) {

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

} else {

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

}

}

}

return dp[n][capacity];

}

public static void main(String[] args) { int[] profits = {15, 25, 13, 23};

int[] weights = {2, 6, 12, 9}; int capacity = 20;

System.out.println("Maximum profit: " + knapsack(profits, weights, capacity));

}

}

# Sample Input and Output:

* Input: Profits: {15, 25, 13, 23}, Weights: {2, 6, 12, 9}, Capacity: 20
* Output: Maximum profit: 48

# Problem 10: Tower of Hanoi Problem

**Illustration:**

The Tower of Hanoi problem involves moving a stack of disks from one peg to another, using a third peg as auxiliary, under the constraints that only one disk can be moved at a time and no disk may be placed on top of a smaller disk.

Example: Given:

* Number of disks *N*=3.
* Pegs: A (source), B (auxiliary), C (destination).

Step-by-Step Solution:

For N=3, move disks from peg A to peg C using peg B.

* + 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.

# Algorithm:

1. Move the top n-1 disks from the source peg to the auxiliary peg.
2. Move the nth disk from the source peg to the destination peg.
3. Move the n-1 disks from the auxiliary peg to the destination peg.

# Source Code:

public class TowerOfHanoi {

public static void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod) { if (n == 1) {

System.out.println("Move disk 1 from rod " + from\_rod + " to rod " + to\_rod); return;

}

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

System.out.println("Move disk " + n + " from rod " + from\_rod + " to rod " + to\_rod); towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

public static void main(String[] args) { int n = 3; // Number of disks towerOfHanoi(n, 'A', 'C', 'B');

}

}

# Sample Input and Output:

* Input: n = 3
* Output:

Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C