

**PABNA UNIVRSITY OF SCIENCE AND TECHNOLOGY**

**Department of Information and Communication Engineering (ICE)**

**LAB REPORT**

**Course Code: ICE-2202**

**Course Title: Data Structure and Algorithm Sessional**

|  |  |
| --- | --- |
| Submitted by:  **Sazia Afrin Labonna**  Roll: 210621  Session: 2020-21  2nd Year 2ndSemester  Department of ICE, PUST | Submitted to:  **Md. Anwar Hossain**  Professor  Department of Information and Communication Engineering(ICE),  Pabna University of Science and Technology, Pabna. |

Date of Submission: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 a linear array using the bubble sort algorithm.

**Illustration of the problem:**

**Bubble Sort Logic Explanation:**

1. **Bubble Sort Algorithm**:

* Traverse through the entire array.
* Compare each pair of adjacent elements, and if the current element is larger than the next, swap them.
* After each full pass, the largest unsorted element is in its correct position, hence "bubbling" up to the end of the array.
* Repeat the process for the remaining unsorted portion of the array.

1. **Illustration**:

* At each pass through the array, you will see the largest unsorted element move to its correct position.
* This helps illustrate the iterative nature of Bubble Sort

**C++ Program with Illustration:**

#include <iostream>

using namespace std;

// Function to perform Bubble Sort

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

bool swapped; // Flag to check if any swap occurs

// Traverse through all array elements

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

swapped = false;

// Print the array state before each pass

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

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

cout << arr[j] << " ";

}

cout << endl;

// Last i elements are already sorted, so no need to check them

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

// Swap if the element found is greater than the next element

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

// Swap elements

int temp = arr[j];

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

arr[j + 1] = temp;

swapped = true;

}

}

// If no two elements were swapped in the inner loop, then the array is already sorted

if (!swapped) {

break;

}

}

}

// Function to print an array

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

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

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

}

cout << endl;

}

int main() {

// Define an example array

int arr[] = {64, 34, 25, 12, 22, 11, 90};

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

cout << "Original array: ";

printArray(arr, n);

// Perform Bubble Sort

bubbleSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Explanation of the Program:**

**bubbleSort function**:

* The function sorts the array using the Bubble Sort algorithm.
* It includes an optimization: If during any pass no elements are swapped, the algorithm stops early because the array is already sorted.

**Print the array state**:

* After every pass through the array, we print the current state of the array to illustrate the sorting process.

**Example with Illustrations:**

Let's walk through the program with an example array {64, 34, 25, 12, 22, 11, 90}:

**Original Array:**

64 34 25 12 22 11 90

**Pass 1:**

1. Compare 64 and 34, swap them: {34, 64, 25, 12, 22, 11, 90}
2. Compare 64 and 25, swap them: {34, 25, 64, 12, 22, 11, 90}
3. Compare 64 and 12, swap them: {34, 25, 12, 64, 22, 11, 90}
4. Compare 64 and 22, swap them: {34, 25, 12, 22, 64, 11, 90}
5. Compare 64 and 11, swap them: {34, 25, 12, 22, 11, 64, 90}
6. Compare 64 and 90, no swap needed.

State after Pass 1:

34 25 12 22 11 64 90

**Pass 2:**

1. Compare 34 and 25, swap them: {25, 34, 12, 22, 11, 64, 90}
2. Compare 34 and 12, swap them: {25, 12, 34, 22, 11, 64, 90}
3. Compare 34 and 22, swap them: {25, 12, 22, 34, 11, 64, 90}
4. Compare 34 and 11, swap them: {25, 12, 22, 11, 34, 64, 90}
5. Compare 34 and 64, no swap needed.
6. Compare 64 and 90, no swap needed.

State after Pass 2:

25 12 22 11 34 64 90

**Pass 3:**

1. Compare 25 and 12, swap them: {12, 25, 22, 11, 34, 64, 90}
2. Compare 25 and 22, swap them: {12, 22, 25, 11, 34, 64, 90}
3. Compare 25 and 11, swap them: {12, 22, 11, 25, 34, 64, 90}
4. Compare 25 and 34, no swap needed.
5. Compare 34 and 64, no swap needed.
6. Compare 64 and 90, no swap needed.

State after Pass 3:

12 22 11 25 34 64 90

**Pass 4:**

1. Compare 12 and 22, no swap needed.
2. Compare 22 and 11, swap them: {12, 11, 22, 25, 34, 64, 90}
3. Compare 22 and 25, no swap needed.
4. Compare 25 and 34, no swap needed.
5. Compare 34 and 64, no swap needed.
6. Compare 64 and 90, no swap needed.

State after Pass 4:

12 11 22 25 34 64 90

**Pass 5:**

1. Compare 12 and 11, swap them: {11, 12, 22, 25, 34, 64, 90}
2. Compare 12 and 22, no swap needed.
3. Compare 22 and 25, no swap needed.
4. Compare 25 and 34, no swap needed.
5. Compare 34 and 64, no swap needed.
6. Compare 64 and 90, no swap needed.

State after Pass 5:

11 12 22 25 34 64 90

**Final Sorted Array:**

11 12 22 25 34 64 90

Top of Form

Bottom of Form

**Algorithm:**

Here is the step-by-step algorithm for sorting a linear array using the Bubble Sort technique:

**1.**Start with the first element of the array.

**2.**Compare each element with the next element in the array.

**3.**Swap the elements if the current element is greater than the next element.

**4.**Move to the next element and repeat the comparison and swapping steps.

**5.**After each complete pass through the array, the largest unsorted element "bubbles up" to its correct position.

**6.**Repeat the process for all elements, reducing the number of comparisons in each subsequent pass, as the largest elements are now sorted.

**7.**Stop when no more swaps are needed, indicating that the array is sorted.

**Source Code:**

#include <iostream>

using namespace std;

int main() {

int n;

// Input for array size

cout << "Enter the size of the array: ";

cin >> n;

int arr[n];

// Input for array elements

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

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

cin >> arr[i];

}

// Bubble sort with exactly 5 passes

for (int pass = 1; pass <= 5; pass++) {

cout << "\nPass " << pass << ":" << endl;

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

// Compare and swap if necessary

cout << "Comparing " << arr[i] << " and " << arr[i + 1] << " -> ";

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

int temp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = temp;

cout << "Swapped";

} else {

cout << "Not swapped";

}

cout << endl;

}

// Print array after each pass

cout << "Array after pass " << pass << ": ";

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

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

}

cout << endl;

}

// Final sorted array

cout << "\nFinal sorted array after 5 passes: ";

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

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

}

cout << endl;

return 0;

}

**Input:**

Enter the size of the array: 5

Enter 5 elements of the array: 3 2 6 1 5

**Output:**

Pass 1:

Comparing 3 and 2 -> Swapped

Comparing 3 and 6 -> Not swapped

Comparing 6 and 1 -> Swapped

Comparing 6 and 5 -> Swapped

Array after pass 1: 2 3 1 5 6

Pass 2:

Comparing 2 and 3 -> Not swapped

Comparing 3 and 1 -> Swapped

Comparing 3 and 5 -> Not swapped

Array after pass 2: 2 1 3 5 6

Pass 3:

Comparing 2 and 1 -> Swapped

Comparing 2 and 3 -> Not swapped

Array after pass 3: 1 2 3 5 6

Pass 4:

Comparing 1 and 2 -> Not swapped

Array after pass 4: 1 2 3 5 6

Pass 5:

Array after pass 5: 1 2 3 5 6

Final sorted array after 5 passes: 1 2 3 5 6

**Problem No:** 02

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

**Illustration of the problem:**

**Problem Explanation:**

You are given an array (or list) of numbers, and you need to search for a specific element (target) in that array. The goal is to check if the target element exists in the array and if so, return the index where the element is found. If the target element is not found, return -1 to indicate that the element does not exist in the array.

**Linear Search Algorithm (Logic):**

1.Start from the first element of the array.

2.Compare the current element with the target element.

3.If the current element matches the target:

Return the index where the target was found.

4.If the current element does not match the target, move to the next element and repeat the comparison.

5.If you reach the end of the array without finding the target, return -1 to indicate the element was not found.

**Key Points:**

Linear Search works by checking each element sequentially.

It is a simple and easy-to-understand algorithm, but it is not efficient for large datasets because it can take up to O(n) time in the worst case, where n is the number of elements in the array.

**Illustration of the Problem:**

Suppose we have the following array:

arr = [10, 20, 30, 40, 50]

And we are searching for the target element 30.

**Steps:**

Start at index 0: Check arr[0] (which is 10). It is not equal to 30.

Move to index 1: Check arr[1] (which is 20). It is not equal to 30.

Move to index 2: Check arr[2] (which is 30). It matches the target.

Return index 2 where the target is found.

If the target was not found after checking all elements, we would return -1.

### **Algorithm:**

### **Step-by-Step Algorithm for Linear Search**

1. **Initialize**:
   * Start with an array of elements and the target value you want to search for.
   * Set a flag variable (e.g., found) to indicate if the element is found.
2. **Iterate through the array**:
   * Loop through each element in the array.
   * Compare each element with the target value.
3. **Check for match**:
   * If the current element matches the target value:
     + Record its index (or position).
     + Set found to true.
     + Exit the loop.
   * Otherwise, continue to the next element.
4. **Output result**:
   * If found is true, print the index where the element was found.
   * If not, indicate that the element is not in the array.

**Source Code:**

#include <iostream>

using namespace std;

// Function to perform linear search

int linearSearch(int arr[], int size, int target) {

// Traverse through the array to find the target element

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

if (arr[i] == target) {

return i;  // Element found, return index

}

}

return -1;  // Element not found, return -1

}

int main() {

int size, target;

// Take the size of the array from the user

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

cin >> size;

// Declare an array of given size

int arr[size];

// Take input for array elements from the user

cout << "Enter " << size << " elements: ";

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

cin >> arr[i];

}

// Take the target element to search for

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

cin >> target;

// Call the linearSearch function

int result = linearSearch(arr, size, target);

// Output the result

if (result != -1) {

cout << "Element " << target << " found at index " << result << ".\n";

} else {

cout << "Element " << target << " not found in the array.\n";

}

return 0;

}

**Input:**

Enter the number of elements in the array: 6

Enter 6 elements: 1 5 3 7 9 11

Enter the element to search for: 7

**Output:**

Element 7 found at index 3.

**Problem No:** 03

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

**Illustration of the problem:**

**Problem Description:**

The goal is to sort an array of integers using the **Merge Sort** algorithm. Merge Sort is a divide-and-conquer algorithm, meaning it divides the problem into smaller subproblems, solves them, and then combines the results to solve the original problem.

**Merge Sort Algorithm:**

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort both halves.
3. **Combine**: Merge the two sorted halves to get the final sorted array.

**Step-by-Step Illustration:**

Let’s consider an example to understand the logic behind Merge Sort.

Example: Array: [38, 27, 43, 3, 9, 82, 10]

1. **Initial Array**: [38, 27, 43, 3, 9, 82, 10]
   * **Step 1**: Split the array into two halves:
     + Left half: [38, 27, 43]
     + Right half: [3, 9, 82, 10]
2. **Sort Left Half [38, 27, 43]**:
   * Split [38, 27, 43] into two halves:
     + Left: [38]
     + Right: [27, 43]
   * [38] is already sorted as it has one element.
   * **Sort Right Half [27, 43]**:
     + Split [27, 43] into two halves:
       - Left: [27]
       - Right: [43]
     + Both [27] and [43] are sorted by default.
     + Now, merge [27] and [43]:
       - Compare 27 and 43, and since 27 < 43, the merged result is [27, 43].
   * Now, merge [38] and [27, 43]:
     + Compare 38 with 27 → place 27 first.
     + Compare 38 with 43 → place 38 next.
     + The merged result is [27, 38, 43].
3. **Sort Right Half [3, 9, 82, 10]**:
   * Split [3, 9, 82, 10] into two halves:
     + Left: [3, 9]
     + Right: [82, 10]
   * **Sort Left Half [3, 9]**:
     + Split [3, 9] into two halves:
       - Left: [3]
       - Right: [9]
     + Both [3] and [9] are sorted by default.
     + Now, merge [3] and [9]:
       - Compare 3 and 9 → the merged result is [3, 9].
   * **Sort Right Half [82, 10]**:
     + Split [82, 10] into two halves:
       - Left: [82]
       - Right: [10]
     + Both [82] and [10] are sorted by default.
     + Now, merge [82] and [10]:
       - Compare 82 and 10 → place 10 first, followed by 82.
       - The merged result is [10, 82].
   * Now, merge [3, 9] and [10, 82]:
     + Compare 3 with 10 → place 3 first.
     + Compare 9 with 10 → place 9 next.
     + Compare 82 with the remaining element 10 → place 10 next.
     + Place 82 last.
     + The merged result is [3, 9, 10, 82].
4. **Merge the Sorted Left Half [27, 38, 43] and Right Half [3, 9, 10, 82]**:
   * Compare 27 with 3 → place 3 first.
   * Compare 27 with 9 → place 9 next.
   * Compare 27 with 10 → place 10 next.
   * Compare 27 with 82 → place 27 next.
   * Compare 38 with 82 → place 38 next.
   * Compare 43 with 82 → place 43 next.
   * Place 82 last.
   * The final sorted array is [3, 9, 10, 27, 38, 43, 82].

**Logic:**

1. **Divide** the array into two halves until each subarray has only one element (base case).
2. **Conquer** by recursively sorting the two halves.
3. **Combine** by merging the two sorted halves into one sorted array.

**Algorithm:**

Merge Sort is a **divide-and-conquer** algorithm used to sort arrays efficiently. The key steps are:

1. **Divide:** Split the array into two halves until each sub-array contains only one element.
2. **Conquer:** Merge the two sorted halves into a single sorted array.
3. **Combine:** Repeat merging sub-arrays step by step until you rebuild the fully sorted array.

**Steps in Detail:**

1. **Recursive Division:**
   * If the array has more than one element:
     + Find the middle index of the array.
     + Divide the array into left and right sub-arrays.
     + Recursively sort both sub-arrays.
2. **Merge Sub-arrays:**
   * Compare elements from the two sub-arrays and merge them into a sorted array.
   * Continue merging until all elements are in sorted order.

**Source Code:**

#include <iostream>

using namespace std;

// Function to merge two halves of the array

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

int n1 = mid - left + 1;  // Size of the left subarray

int n2 = right - mid;     // Size of the right subarray

// Create temporary arrays for left and right subarrays

int leftArr[n1], rightArr[n2];

// Copy data into temporary arrays

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

leftArr[i] = arr[left + i];

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

rightArr[i] = arr[mid + 1 + i];

// Merge the temporary arrays back into the original array

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

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

// Copy any remaining elements of leftArr

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

// Copy any remaining elements of rightArr

while (j < n2) {

arr[k] = rightArr[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;

// Recursively sort the two halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Function to print the array

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

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

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

cout << endl;

}

int main() {

int n;

// Input the number of elements

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

cin >> n;

int arr[n];

// Input the array elements

cout << "Enter " << n << " elements: ";

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

cin >> arr[i];

}

// Perform merge sort

mergeSort(arr, 0, n - 1);

// Output the sorted array

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Input:**

Enter the number of elements: 5

Enter 5 elements: 38 27 43 3 9

**Output:**

Sorted array: 3 9 27 38 43

**Problem No:** 04

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

**Illustration of the problem:**

**Problem Statement:**

We are given a **sorted array** and a **target element**. Our goal is to find the index of the target element in the array using the **Binary Search** algorithm.

**Binary Search Algorithm:**

Binary Search works by dividing the sorted array into two halves repeatedly, and then checking whether the target element lies in the left or right half. This halving continues until the target element is found or the search space is exhausted.

**Logic Behind Binary Search:**

1. **Divide the Array**: Start with the entire array.
2. **Find the Middle Element**: Calculate the middle index of the array.
3. **Compare**:
   * If the middle element is equal to the target element, return the index of the middle element (target found).
   * If the target element is less than the middle element, search the left half of the array.
   * If the target element is greater than the middle element, search the right half of the array.
4. **Repeat** the above steps for the left or right subarray, halving the array size each time, until the target element is found or the subarray becomes empty (indicating the target is not in the array).

**Illustration:**

Consider the sorted array:

[10, 20, 30, 40, 50, 60, 70, 80, 90]

Let’s search for the target value **50** using Binary Search.

**Step-by-Step Process:**

1. **Initial Array**: [10, 20, 30, 40, 50, 60, 70, 80, 90]
   * Left index = 0, Right index = 8
   * Middle index = (0 + 8) / 2 = 4
   * Middle element = arr[4] = 50
   * The middle element is equal to the target (50), so we return index 4.

In this case, the binary search was able to find the element on the first try, but let’s consider a different example for further understanding.

**Example 2: Searching for 70**

1. **Initial Array**: [10, 20, 30, 40, 50, 60, 70, 80, 90]
   * Left index = 0, Right index = 8
   * Middle index = (0 + 8) / 2 = 4
   * Middle element = arr[4] = 50
   * The target 70 is greater than 50, so we discard the left half and search the right half: New left index = 5, right index = 8.
2. **New Subarray**: [60, 70, 80, 90]
   * Left index = 5, Right index = 8
   * Middle index = (5 + 8) / 2 = 6
   * Middle element = arr[6] = 70
   * The middle element is equal to the target (70), so we return index 6.

Thus, **70** is found at index 6.

**Example 3: Searching for 25**

1. **Initial Array**: [10, 20, 30, 40, 50, 60, 70, 80, 90]
   * Left index = 0, Right index = 8
   * Middle index = (0 + 8) / 2 = 4
   * Middle element = arr[4] = 50
   * The target 25 is smaller than 50, so we discard the right half and search the left half: New left index = 0, right index = 3.
2. **New Subarray**: [10, 20, 30, 40]
   * Left index = 0, Right index = 3
   * Middle index = (0 + 3) / 2 = 1
   * Middle element = arr[1] = 20
   * The target 25 is greater than 20, so we discard the left half and search the right half: New left index = 2, right index = 3.
3. **New Subarray**: [30, 40]
   * Left index = 2, Right index = 3
   * Middle index = (2 + 3) / 2 = 2
   * Middle element = arr[2] = 30
   * The target 25 is smaller than 30, so we discard the right half and search the left half: New left index = 2, right index = 1.
4. **Left index is greater than Right index**, indicating the target element **25** is not in the array.

**Algorithm:**

**Binary Search Algorithm: Step-by-Step Explanation**

**Step-by-Step Algorithm:**

**Input:**

* A sorted array arr[] of n elements.
* A **target element** target to search for in the array.

**Output:**

* If the **target** is found, return its index.
* If the **target** is not found, return -1.

**Binary Search Steps:**

1. **Initialize pointers**:
   * Set left = 0 (the index of the first element).
   * Set right = n - 1 (the index of the last element).
2. **Repeat until left is less than or equal to right**:
   * Calculate the **middle index** of the current subarray:

mid = left + (right - left) / 2

This formula helps avoid overflow in case the sum of left and right becomes too large.

1. **Check the middle element**:
   * If arr[mid] == target, then the target is found at index mid. **Return mid**.
   * If arr[mid] < target, the target must be in the right half of the array. Thus, update left to mid + 1.
   * If arr[mid] > target, the target must be in the left half of the array. Thus, update right to mid - 1.
2. **Repeat**:
   * Keep narrowing down the search range by updating the values of left and right according to the above conditions.
   * Continue until either the target is found (i.e., arr[mid] == target) or the subarray becomes invalid (left > right).
3. **If the loop ends**:
   * If left > right, it means the target is not in the array. **Return -1**.

**Source Code:**

#include <iostream>

#include <algorithm>  // For std::sort

using namespace std;

// Function to perform binary search on a sorted array

int binarySearch(int arr[], int n, int target) {

int left = 0;

int right = n - 1;

// Perform binary search

while (left <= right) {

int mid = left + (right - left) / 2; // Find the middle index

// If the target element is found at the mid index

if (arr[mid] == target) {

return mid; // Return the index of the element

}

// If the target is greater than the mid element, ignore the left half

else if (arr[mid] < target) {

left = mid + 1;

}

// If the target is smaller than the mid element, ignore the right half

else {

right = mid - 1;

}

}

// Return -1 if the target element is not found

return -1;

}

int main() {

int n, target;

// Input: number of elements in the array

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

cin >> n;

int arr[n];

// Input: elements of the array

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

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

cin >> arr[i];

}

// Sorting the array before applying binary search (Binary Search requires a sorted array)

sort(arr, arr + n);

// Input: the element to search

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

cin >> target;

// Perform binary search

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

// Output: result of the search

if (result != -1) {

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

} else {

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

}

return 0;

}

**Input:**

Enter the number of elements: 5

Enter the elements of the array: 50 10 30 20 40

Enter the element to search: 30

**Output:**

Element found at index: 2

**Problem No:** 05

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

**Illustration of the problem:**

**Problem Illustration: Pattern Matching**

Given:

* **Text**: A string of characters (can be long).
* **Pattern**: A string that we want to find in the **text**.

We need to find all occurrences of the **pattern** in the **text**.

**Example:**

**Text**: "ababcabcababc" **Pattern**: "abc"

We need to find all positions where "abc" appears in the **text**.

**Approach to Solution:**

**Naive Pattern Matching Algorithm**

The Naive Pattern Matching algorithm tries to match the pattern starting from every position in the text. At each position, it checks if the characters from that position match the pattern. If a complete match is found, it prints the position. Here's how this works step-by-step:

1. Start from the beginning of the text.
2. For each position in the text where the pattern can possibly fit (i.e., i from 0 to n-m where n is the length of text and m is the length of pattern):
   * Compare the pattern with the substring of text starting from the current position i.
   * If all characters match, print the index.
3. Repeat until all positions in the text are checked.

**Logic:**

* If the pattern is found at position i, the program will print that position.
* The comparison is done by iterating through the text and checking if characters match sequentially.

**Algorithm:**

**Naive Pattern Matching Algorithm**

**Problem:**

Given a text string and a pattern string, the goal is to find all the occurrences of the pattern in the text.

**Algorithm:**

1. **Input**:
   * Text (a string of length n).
   * Pattern (a string of length m).
2. **Output**:
   * The starting index (0-based) of all occurrences of Pattern in Text. If no match is found, indicate that the pattern doesn't exist in the text.

**Steps:**

1. **Initialize Variables**:
   * Let n be the length of Text.
   * Let m be the length of Pattern.
2. **Iterate through the Text**:
   * For each position i (from 0 to n - m), do the following:
     + Let j = 0.
     + **Inner Loop**: Compare the substring of Text starting at index i with the Pattern. Do this by checking:
       - If Text[i + j] == Pattern[j] for each j from 0 to m - 1 (i.e., compare character by character).
       - If Text[i + j] != Pattern[j] for any j, break out of the loop and move to the next position in the text (i + 1).
3. **Pattern Match Check**:
   * If the loop completes and j == m (i.e., all characters of the pattern have matched), print the index i (i.e., Pattern is found at position i in Text).
4. **Return**:
   * If no matches are found by the end of the loop, return that the pattern is not found.

**Source Code:**

#include <iostream>

#include <string>

using namespace std;

// Function to implement the Naive String Matching Algorithm

void naivePatternSearch(const string& text, const string& pattern) {

int n = text.length();

int m = pattern.length();

bool found = false;

// Loop through the text

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

int j = 0;

// Check for the pattern at the current position

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

j++;

}

// If all characters of the pattern are matched

if (j == m) {

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

found = true;

}

}

// If no match is found

if (!found) {

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

}

}

int main() {

string text, pattern;

// Taking input from the user

cout << "Enter the text: ";

getline(cin, text);

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

getline(cin, pattern);

// Call the Naive pattern search function

naivePatternSearch(text, pattern);

return 0;

}

**Input & Output:**

Enter the text: Hello, welcome to the world of C++ programming.

Enter the pattern to search for: welcome

Pattern found at index 7

Enter the text: This is a test string.

Enter the pattern to search for: C++

Pattern not found in the text.

**Problem No:** 06

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

**Illustration of the problem:**

**Queue Data Structure Overview:**

A **Queue** is a linear data structure that follows the **First-In, First-Out (FIFO)** principle. This means:

* The element that enters the queue **first** will be the **first** one to be removed.
* Think of a line at a ticket counter: the first person in line is the first to get a ticket and leave.

**Basic Queue Operations:**

1. **Enqueue** (Add an element to the queue)
2. **Dequeue** (Remove an element from the queue)
3. **Peek** (View the front element without removing it)
4. **isFull** (Check if the queue is full)
5. **isEmpty** (Check if the queue is empty)

**Illustration of Queue Operations:**

Let's imagine a queue of size 5. We'll visualize the following operations step-by-step.

**Initial State (Queue is empty):**

Queue: [ - , - , - , - , - ] <-- All positions are empty

Front: 0, Rear: -1, Size: 0

Here, the queue is empty, and no elements have been added.

**1. Enqueue Operation (Add elements to the queue)**

Let's **enqueue** values 10, 20, and 30 to the queue.

* After adding 10:

Queue: [ 10, - , - , - , - ] <-- 10 is added at the rear

Front: 0, Rear: 0, Size: 1

* After adding 20:

Queue: [ 10, 20, - , - , - ] <-- 20 is added at the rear

Front: 0, Rear: 1, Size: 2

* After adding 30:

Queue: [ 10, 20, 30, - , - ] <-- 30 is added at the rear

Front: 0, Rear: 2, Size: 3

Now, the queue has 3 elements, and the front is at index 0, while the rear is at index 2.

**2. Dequeue Operation (Remove elements from the queue)**

Let's **dequeue** an element.

* Removing the element at the front (which is 10):

Queue: [ - , 20, 30, - , - ] <-- 10 is removed from the front

Front: 1, Rear: 2, Size: 2

After dequeuing, the queue's front has moved to index 1, and the element 10 has been removed.

**3. Enqueue Again**

Let's now add another element, 40.

* After adding 40:

Queue: [ - , 20, 30, 40, - ] <-- 40 is added at the rear

Front: 1, Rear: 3, Size: 3

The rear moves to index 3, and the queue now contains 20, 30, and 40.

**4. Dequeue Again**

Next, let's dequeue the element at the front, which is 20.

* After removing 20:

Queue: [ - , - , 30, 40, - ] <-- 20 is removed from the front

Front: 2, Rear: 3, Size: 2

The front has moved to index 2, and the element 20 is removed.

**5. Peek Operation (View the front element)**

Now, let's see what the element at the front of the queue is without removing it.

* The **peek** operation would show 30 (the front element).

Front: 2, Rear: 3, Size: 2

Peek: 30

**6. Queue After Some More Operations (Full Queue)**

Let's continue adding more elements to the queue until it reaches full capacity.

* Add 50:

Queue: [ - , - , 30, 40, 50 ] <-- 50 is added at the rear

Front: 2, Rear: 4, Size: 3

At this point, the queue has elements 30, 40, and 50. We have a total size of 3.

* Finally, if we add 60:

Queue: [ 60, - , 30, 40, 50 ] <-- 60 is added at the front (because it’s a circular queue)

Front: 3, Rear: 0, Size: 4

**Full Queue Illustration (Wrap Around):**

With a **circular queue** mechanism, when the rear reaches the end of the array, the next enqueue operation wraps around to the start of the array.

For example, if the queue size is 5, and the rear reaches index 4, the next enqueue would happen at index 0, creating a circular structure.

**Queue Logic Breakdown:**

1. **Enqueue** operation adds an element to the queue by placing it at the rear. If the rear reaches the end of the queue, it wraps around to the start using the formula:
   * rear = (rear + 1) % capacity
2. **Dequeue** operation removes the element from the front of the queue, and the front index is incremented using the formula:
   * front = (front + 1) % capacity
3. **Peek** operation returns the element at the front without modifying the queue, so the front index does not change.
4. **isEmpty** checks if the size of the queue is zero.
5. **isFull** checks if the size of the queue equals its capacity.

**Circular Queue Key Concept:**

* A **circular queue** ensures that the queue doesn’t get stuck when the rear reaches the end of the array. Instead of continuing to increase the rear index, the rear wraps around to the beginning of the array when there's space. This helps maximize space utilization.

**Conclusion:**

The queue data structure follows a simple but efficient FIFO principle. Operations like enqueue and dequeue are carried out in constant time **O(1)**, and the use of a circular array avoids wastage of space. This is ideal for situations like task scheduling, buffer management, and other real-time applications.

**Algorithm:**

Here’s the **algorithm** for implementing a **Queue** data structure with basic operations (Enqueue, Dequeue, Peek, Display) using an array-based approach.

**Queue Algorithm (Array-based)**

**Problem:**

We need to implement a queue using an array, with the operations:

* **Enqueue**: Insert an element into the queue.
* **Dequeue**: Remove an element from the queue.
* **Peek**: View the front element without removing it.
* **isEmpty**: Check if the queue is empty.
* **isFull**: Check if the queue is full.
* **Display**: Show all the elements in the queue.

**Data Structures:**

1. **Array** arr[] to store queue elements.
2. **Integer** front to track the front of the queue.
3. **Integer** rear to track the rear of the queue.
4. **Integer** size to track the current number of elements in the queue.
5. **Integer** capacity to store the maximum capacity of the queue.

**Queue Operations Algorithm:**

1. **Initialization**:

* Create an array arr[] of fixed size capacity.
* Initialize front = 0, rear = -1, size = 0, and capacity to the maximum size of the queue.

**Algorithm for Enqueue Operation:**

1. **Check if the queue is full**:

* If size == capacity, the queue is full. Print an error message: "Queue is full! Cannot enqueue."
* Otherwise, proceed to the next step.

1. **Insert the element at the rear**:

* Increment the rear index: rear = (rear + 1) % capacity (circular increment).
* Add the element at arr[rear].
* Increment the size by 1.

1. **End of operation**.

**Algorithm for Dequeue Operation:**

1. **Check if the queue is empty**:

* If size == 0, the queue is empty. Print an error message: "Queue is empty! Cannot dequeue."
* Otherwise, proceed to the next step.

1. **Remove the element at the front**:

* Store the element at arr[front] and return it.
* Increment the front index: front = (front + 1) % capacity (circular increment).
* Decrease the size by 1.

1. **End of operation**.

**Algorithm for Peek Operation:**

1. **Check if the queue is empty**:

* If size == 0, the queue is empty. Print an error message: "Queue is empty! Cannot peek."
* Otherwise, proceed to the next step.

1. **Return the element at the front**: Return arr[front].
2. **End of operation**.

**Algorithm for Display Operation:**

1. **Check if the queue is empty**:

* If size == 0, the queue is empty. Print an error message: "Queue is empty! Cannot display."
* Otherwise, proceed to the next step.

1. **Display all elements in the queue**:

* Traverse the array from front to rear using circular indexing:
* For each index i from front to rear, print the element arr[i] in the correct order.

1. **End of operation**.

**Algorithm for isEmpty Operation:**

1. **Check if the queue is empty**:

* If size == 0, return true (queue is empty).
* Otherwise, return false.

**Algorithm for isFull Operation:**

1. **Check if the queue is full**:

* If size == capacity, return true (queue is full).
* Otherwise, return false.

**Source Code:**

#include <iostream>

using namespace std;

// Queue class definition

class Queue {

private:

int\* arr;          // Pointer to array that stores queue elements

int front;         // Index of the front element

int rear;          // Index of the rear element

int capacity;      // Maximum size of the queue

int size;          // Current size of the queue

public:

// Constructor to initialize queue

Queue(int cap) {

capacity = cap;

arr = new int[capacity];

front = 0;

rear = -1;

size = 0;

}

// Destructor to free memory

~Queue() {

delete[] arr;

}

// Function to check if the queue is full

bool isFull() {

return size == capacity;

}

// Function to check if the queue is empty

bool isEmpty() {

return size == 0;

}

// Function to add an element to the queue

void enqueue(int value) {

if (isFull()) {

cout << "Queue is full! Cannot enqueue." << endl;

} else {

rear = (rear + 1) % capacity; // Circular increment

arr[rear] = value;

size++;

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

}

}

// Function to remove an element from the queue

int dequeue() {

if (isEmpty()) {

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

return -1; // Indicating an error

} else {

int item = arr[front];

front = (front + 1) % capacity; // Circular increment

size--;

return item;

}

}

// Function to get the front element of the queue

int peek() {

if (isEmpty()) {

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

return -1; // Indicating an error

} else {

return arr[front];

}

}

// Function to display the elements of the queue

void display() {

if (isEmpty()) {

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

} else {

cout << "Queue elements: ";

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

cout << arr[(front + i) % capacity] << " ";

}

cout << endl;

}

}

};

// Main function to test the Queue implementation

int main() {

int capacity, choice, value;

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

cin >> capacity;

// Create a queue with the given capacity

Queue q(capacity);

do {

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

cout << "1. Enqueue\n";

cout << "2. Dequeue\n";

cout << "3. Peek\n";

cout << "4. Display\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

// Enqueue operation

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

cin >> value;

q.enqueue(value);

break;

case 2:

// Dequeue operation

value = q.dequeue();

if (value != -1) {

cout << "Dequeued: " << value << endl;

}

break;

case 3:

// Peek operation

value = q.peek();

if (value != -1) {

cout << "Front element: " << value << endl;

}

break;

case 4:

// Display the queue

q.display();

break;

case 5:

cout << "Exiting the program." << endl;

break;

default:

cout << "Invalid choice, please try again." << endl;

}

} while (choice != 5);

return 0;

}

**Input & Output:**

Enter the capacity of the queue: 5

Queue Operations Menu:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Exit

Enter your choice: 1

Enter the value to enqueue: 10

10 enqueued to queue.

Queue Operations Menu:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Exit

Enter your choice: 1

Enter the value to enqueue: 20

20 enqueued to queue.

Queue Operations Menu:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Exit

Enter your choice: 4

Queue elements: 10 20

Queue Operations Menu:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Exit

Enter your choice: 2

Dequeued: 10

Queue Operations Menu:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Exit

Enter your choice: 3

Front element: 20

**Problem No:** 07

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

**Illustration of the problem:**

The **n-Queens problem** involves placing **n queens** on an **n x n** chessboard such that no two queens threaten each other. A queen can attack another queen if they share the same row, column, or diagonal.

**Backtracking Approach:**

To solve the n-Queens problem using backtracking:

1. **Start with an empty chessboard**.
2. **Place a queen in a valid position** in the current row.
3. **Check if the position is safe** — no other queens should threaten it.
4. **Recursively place queens** on the next rows.
5. **Backtrack** if placing a queen leads to an invalid configuration.

The backtracking algorithm will try placing queens one by one in each row. If it reaches a row where no valid position exists, it will backtrack to the previous row and try a different position.

**Steps to Illustrate the Logic:**

1. For each row, place a queen in each column.
2. After placing a queen, check for safety (i.e., no other queens are in the same column, same diagonal, or same row).
3. If placing a queen in a column does not result in a conflict, recursively attempt to place queens in the next row.
4. If we cannot place a queen in any column of the current row, backtrack and move the queen to the next column in the previous row.

**Algorithm:**

1. **Start** with row = 0.
2. **For each column** (from 0 to n-1) in the current row:

* **Check if placing a queen** in (row, col) is safe using the isSafe function.
* If safe:
* Place the queen (i.e., mark board[row][col] = 1).
* **Recurse** to the next row (row + 1).
* If placing the queen leads to a valid solution, return true.
* **Backtrack**: If no valid solution is found, remove the queen (board[row][col] = 0) and try the next column.

1. **If all rows** are filled successfully, return true (solution found).
2. **If no placement** is possible in a row, backtrack to the previous row.
3. If all rows are attempted and no solution is found, return false (no solution).

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

bool isSafe(const vector<vector<int>>& board, int row, int col, int n) {

// Check the column

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

if (board[i][col] == 1) {

return false;

}

}

// Check the diagonal (top-left to bottom-right)

for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 1) {

return false;

}

}

// Check the diagonal (top-right to bottom-left)

for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {

if (board[i][j] == 1) {

return false;

}

}

return true;

}

bool solveNQueens(vector<vector<int>>& board, int row, int n) {

// If all queens are placed

if (row == n) {

return true;

}

// Try placing the queen in each column of the current row

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

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

board[row][col] = 1;  // Place the queen

// Recursively place queens in the next row

if (solveNQueens(board, row + 1, n)) {

return true;

}

// If placing queen in the current position doesn't lead to a solution,

// backtrack (remove the queen)

board[row][col] = 0;

}

}

return false;  // If no valid position is found, return false

}

void printBoard(const vector<vector<int>>& board, int n) {

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

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

if (board[i][j] == 1)

cout << "Q ";  // Queen represented by 'Q'

else

cout << ". ";  // Empty space represented by '.'

}

cout << endl;

}

}

int main() {

int n;

// Ask user for the value of n (number of queens)

cout << "Enter the number of queens (n): ";

cin >> n;

// Initialize an n x n board with all values set to 0

vector<vector<int>> board(n, vector<int>(n, 0));

// Try to solve the N-Queens problem

if (solveNQueens(board, 0, n)) {

cout << "Solution found:\n";

printBoard(board, n);

} else {

cout << "No solution exists for n = " << n << endl;

}

return 0;

}

**Input:**

Enter the number of queens: 4

**Output:**

Solution found:

Q . . .

. . . Q

. Q . .

. . Q .

**Problem No:** 08

**Title:** Consider a set S={5,10,12,13,15,18} and d=30. Write a program to solve the sum of subset problem.

**Illustration of the problem:**

The Sum of Subsets problem is a well-known combinatorial problem where, given a set of numbers and a target sum, you need to find all subsets of the set whose elements sum up to a target value. In your case, the set S={5,10,12,13,15,18}S = \{5, 10, 12, 13, 15, 18\}S={5,10,12,13,15,18} and the target sum d=30d = 30d=30.

**Problem Breakdown:**

You need to find all subsets of the set SSS that sum up to d=30d = 30d=30. Each subset is a combination of the elements from SSS, and you are tasked with finding those combinations that add up to exactly 30.

**Approach:**

To solve this problem, we can use **backtracking**. This approach will explore all possible subsets and check if their sum matches the target. The idea is to recursively decide whether to include an element in the current subset or not.

**Logic of the Solution:**

1. **Backtracking/Recursion**: Start from the first element and try including it in the current subset. If including it doesn’t exceed the sum, continue to the next element. If the sum exceeds the target, backtrack.
2. **Base Case**: If the sum of the current subset equals the target ddd, we have found a valid subset.
3. **Pruning**: If the sum exceeds the target, stop exploring further.

**Algorithm:**

1. **Input**:

* A set of numbers S={s1,s2,...,sn}S = \{s\_1, s\_2, ..., s\_n\}S={s1​,s2​,...,sn​}
* A target sum ddd

1. **Initialization**:

* Start with an empty subset.
* Set the current sum of the subset to 0.
* Set the index to 0 (the first element of the set).

1. **Recursive Backtracking**:

* For each element in the set, do the following:
  + - **Include the current element**:
      * Add the current element to the subset.
      * Update the current sum by adding the current element’s value.
      * Recursively call the function with the next element.
    - **Exclude the current element**:
      * Remove the current element from the subset (backtrack).
      * Recursively call the function without adding the current element to the subset.

1. **Base Case**:
   * If the sum of the current subset equals the target sum ddd, print the current subset.
   * If the current sum exceeds ddd or the index is out of bounds (i.e., all elements have been considered), stop further exploration along that path.
2. **End Condition**:
   * After all recursive calls are made, the program will have printed all subsets whose sum equals the target sum.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

// Function to find the sum of subsets

void findSubsets(vector<int>& set, vector<int>& subset, int index, int sum, int d) {

// If the sum is equal to d, print the subset

if (sum == d) {

cout << "{ ";

for (int num : subset) {

cout << num << " ";

}

cout << "}" << endl;

return;

}

// If we've gone through all elements, return

if (index == set.size()) {

return;

}

// Include the current element in the subset

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

findSubsets(set, subset, index + 1, sum + set[index], d);

// Exclude the current element from the subset

subset.pop\_back();

findSubsets(set, subset, index + 1, sum, d);

}

int main() {

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

int d = 30;  // Desired sum

vector<int> subset;  // This will store the current subset

cout << "Subsets of set whose sum is " << d << " are: " << endl;

findSubsets(set, subset, 0, 0, d);  // Start from index 0 with sum 0

return 0;

}

**Input & Output:**

Subsets of set whose sum is 30 are:

{ 5 10 15 }

{ 12 18 }

{ 5 12 13 }

**Problem No:** 09

**Title:** 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.

**Illustration of the problem:**

To solve the 0/1 Knapsack problem using dynamic programming, the goal is to determine the maximum profit that can be achieved by selecting a subset of items such that their total weight does not exceed the capacity of the knapsack.

**Problem Illustration**

Given:

* Profits: P = {15, 25, 13, 23}
* Weights: W = {2, 6, 12, 9}
* Knapsack Capacity C = 20
* Number of Items n = 4

**Dynamic Programming Approach**

The approach is based on building a table where the cell at dp[i][j] represents the maximum profit achievable with the first i items and a knapsack capacity j.

* **Base Case**: If we have zero items (i.e., i = 0), the maximum profit is 0 for any capacity j (i.e., dp[0][j] = 0).
* **Recurrence Relation**:
  + If we don't include the item i: dp[i][j] = dp[i-1][j]
  + If we include the item i, the profit becomes P[i-1] + dp[i-1][j - W[i-1]] (but we only include it if the current weight does not exceed the capacity).
  + So, dp[i][j] = max(dp[i-1][j], P[i-1] + dp[i-1][j-W[i-1]]) if the item can be included.

**Solution Steps:**

1. Initialize a table dp of size (n+1) x (C+1) (with all values initially set to 0).
2. Fill the table using the above recurrence relation.
3. The value at dp[n][C] will give the maximum profit achievable.

**Algorithm:**

**Algorithm for Solving the 0/1 Knapsack Problem Using Dynamic Programming**

The **0/1 Knapsack Problem** involves selecting a subset of items such that the total weight does not exceed a given capacity, while the total profit is maximized. This problem can be solved using dynamic programming.

**Algorithm Steps:**

**Inputs:**

* **Profits array (P)**: List of profits corresponding to each item.
* **Weights array (W)**: List of weights corresponding to each item.
* **Capacity (C)**: Maximum weight the knapsack can carry.
* **n**: Number of items.

**Step-by-Step Procedure:**

1. **Initialize a 2D DP Table**:
   * Let dp[i][j] represent the maximum profit that can be achieved by considering the first i items with a knapsack capacity of j.
   * The table will be of size (n + 1) x (C + 1), where:
     + Rows represent the number of items (0 to n).
     + Columns represent possible knapsack capacities (0 to C).
2. **Base Case**:
   * If there are no items (i.e., i = 0), or if the capacity is 0 (i.e., j = 0), the maximum profit is 0 because:
     + dp[0][j] = 0 for all j (with 0 items, no profit can be made).
     + dp[i][0] = 0 for all i (with 0 capacity, no item can be added).
3. **Fill the DP Table**:
   * For each item i (from 1 to n):
     + For each capacity j (from 1 to C):
       - If the weight of the current item W[i-1] is less than or equal to the current capacity j, then:
         1. Either include the current item (add its profit to the remaining capacity).
         2. Or exclude it (keep the previous maximum profit).
         3. The recurrence relation is: dp[i][j]=max⁡(dp[i−1][j],P[i−1]+dp[i−1][j−W[i−1]])dp[i][j] = \max(dp[i-1][j], P[i-1] + dp[i-1][j - W[i-1]])
       - If the weight of the current item exceeds the current capacity j, then exclude the item:
         1. dp[i][j] = dp[i-1][j].
4. **Final Solution**:
   * The maximum profit is stored in dp[n][C], which represents the maximum profit achievable with n items and knapsack capacity C.

**Source Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

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

// Create a DP table to store the maximum profit at each weight capacity

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

// Build the table in bottom-up manner

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

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

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

// Max of not taking or taking the item

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

} else {

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

}

}

}

// Return the maximum profit achievable for the knapsack capacity C

return dp[n][C];

}

int main() {

int n, C;

// Take user input for the number of items and knapsack capacity

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

cin >> n;

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

cin >> C;

vector<int> P(n), W(n);

// Take user input for the profits and weights of the items

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

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

cin >> P[i];

}

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

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

cin >> W[i];

}

// Calculate and print the maximum profit

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

cout << "Maximum profit in the knapsack: " << maxProfit << endl;

return 0;

}

**Input:**

Enter the number of items: 4

Enter the capacity of the knapsack: 20

Enter the profits of the items: 15 25 13 23

Enter the weights of the items: 2 6 12 9

**Output:**

Maximum profit in the knapsack: 38

**Problem No:** 10

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

**Illustration of the problem:**

The **Tower of Hanoi** is a classic problem in recursive algorithms. It involves three rods and a number of disks of different sizes. The objective is to move all the disks from the source rod to the target rod, following these rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one of the stacks and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

**Problem Illustration:**

Let's assume we have 3 disks (small, medium, large), and the rods are labeled as A, B, and C.

* The disks are initially placed on rod A in the order from top to bottom: small, medium, large.
* We want to move all disks from rod A to rod C using rod B as an auxiliary.

Here’s how the solution works step-by-step for 3 disks:

1. Move the top 2 disks from rod A to rod B using rod C as auxiliary.
2. Move the largest disk (disk 3) from rod A to rod C.
3. Move the 2 disks from rod B to rod C using rod A as auxiliary.

**General Recursive Approach:**

To solve the problem recursively for n disks:

* **Base case**: If there is only one disk, simply move it from the source rod to the target rod.
* **Recursive case**:
  + Move n-1 disks from the source rod to the auxiliary rod using the target rod.
  + Move the nth disk (largest disk) from the source rod to the target rod.
  + Move the n-1 disks from the auxiliary rod to the target rod using the source rod.

**Algorithm:**

The recursive approach for solving the Tower of Hanoi is as follows:

* Move n-1 disks from the source peg to an auxiliary peg.
* Move the nth disk from the source peg to the destination peg.
* Move the n-1 disks from the auxiliary peg to the destination peg.

This process repeats recursively.

**Source Code:**

#include <iostream>

using namespace std;

// Function to solve the Tower of Hanoi problem

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

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

if (n == 1) {

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

return;

}

// Recursive case:

// Move n-1 disks from source to auxiliary, using destination as auxiliary

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

// Move the nth disk from source to destination

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

// Move n-1 disks from auxiliary to destination, using source as auxiliary

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

}

int main() {

int n;

// Take user input for the number of disks

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

cin >> n;

// Solve the Tower of Hanoi problem for n disks

cout << "The moves are as follows:" << endl;

towerOfHanoi(n, 'A', 'C', 'B');  // A is source, C is destination, B is auxiliary

return 0;

}

**Input:**

Enter the number of disks: 3

**Output:**

The moves are as follows:

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