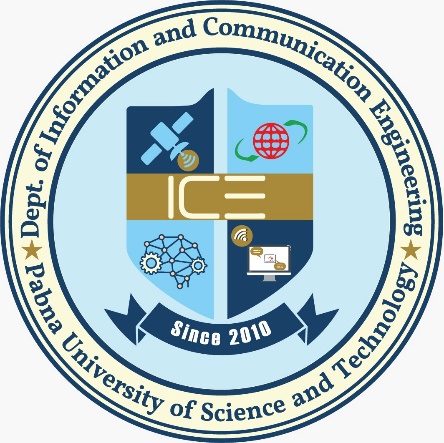


|  |  |
| --- | --- |
| **Sl.** | **Problem Statement** |
| **1.** | |  | | --- | | Write a program to sort a linear array us the bubble sort algorithm. | |
| **2.** | |  | | --- | | Write a program to find an element using a linear search algorithm. | |
| **3.** | |  | | --- | | Write a program to sort 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**

****

**Lab-1:** **Write a program to sort a linear array using the bubble sort algorithm**

**Objectives:**

* Implement the **Bubble Sort** algorithm.
* Display the sorting process after each swap.
* Sort an array in **ascending order**.
* Show the final sorted array.

**Theory:**

* **Bubble Sort** is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order.
* This process is repeated for every element in the array until the entire array is sorted.
* The algorithm consists of multiple passes, where in each pass, the largest unsorted element "bubbles" up to its correct position.

**Time and Space Complexity:**

* **Worst-case time complexity**: O(n^2) (when the array is in reverse order).
* **Best-case time complexity**: O(n) (when the array is already sorted).
* **Average-case time complexity**: O(n^2).
* **Space complexity**: O(1) (in-place sorting, requires no extra space).

**Algorithm (Step by Step):**

1. **Input** the size of the array n.
2. Read n elements into the array.
3. Initialize a counter variable cnt = 1 for tracking passes.
4. Use a **while loop** to iterate through passes until cnt < n:
   * In each pass, iterate through the array using a **for loop**.
   * Compare adjacent elements and swap them if necessary.
   * After every swap, print the current state of the array.
   * Increment cnt after each pass.
5. Print the final sorted array.

**Code:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cout << "Enter n:";

cin >> n;

int a[n];

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

{

cin >> a[i];

}

int cnt = 1;

while (cnt < n)

{

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

{

if (a[i] > a[i + 1])

{

int temp = a[i];

a[i] = a[i + 1];

a[i + 1] = temp;

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

{

cout << a[i] << " ";

}

cout << endl;

}

}

cout << endl;

cnt++;

}

cout << "Final Array is ";

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

{

cout << a[i] << " ";

}

return 0;

}

**Input**

Enter n: 5

5 3 8 4 2

**Output**

3 5 8 4 2

3 5 4 8 2

3 5 4 2 8

3 5 4 2 8

3 4 5 2 8

3 4 2 5 8

3 4 2 5 8

3 2 4 5 8

2 3 4 5 8

Final Array is 2 3 4 5 8

**Lab-2:** **Write a program to find an element using a linear search algorithm.**

**Objectives:**

* To take an array of n elements as input from the user.
* To search for a specific value (v) in the array.
* To determine whether the value is present in the array and display the result.

**Theory:**

This program performs a **linear search** to find whether a given value exists in the array.

* **Linear search** is a simple searching algorithm that checks each element sequentially until a match is found or the entire array is traversed.
* It works efficiently for small datasets but is less efficient for large datasets compared to **binary search**.

**Time and Space Complexity:**

* **Time Complexity:** O(n) (In the worst case, it may have to check all n elements).
* **Space Complexity:** O(1) (Only a few extra variables are used).

**Algorithm (Step by Step):**

1. Take an integer n (size of the array) as input.
2. Declare an array of size n and take its elements as input.
3. Take an integer v as input (the value to search).
4. Initialize flag = 0 (to track if v is found).
5. Iterate through the array:
   * If any element matches v, set flag = 1 and break the loop.
6. After the loop, check the value of flag:
   * If flag == 1, print "Founded".
   * Otherwise, print "Not Founded".

**Code:**

#include <bits/stdc++.h> // Include all standard libraries

using namespace std;

int main() {

int n;

cout << "Enter N: ";

cin >> n; // Take the size of the array as input

int ar[n]; // Declare an array of size n

// Taking array elements as input

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

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

cin >> ar[i];

}

int v;

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

cin >> v; // Take the value to be searched

int flag = 0; // Flag to track if value is found

// Linear search in the array

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

if (ar[i] == v) { // If the value is found

flag = 1;

break; // Stop searching once found

}

}

// Output result

if (flag == 1) {

cout << "Founded" << endl; // If found

} else {

cout << "Not Founded" << endl; // If not found

}

return 0;

}

**Input**

Enter N: 4

Enter 4 elements: 1 2 3 4

Enter the value to search: 10

Output:

Not Founded

**Lab-3: Write a program to sort a linear array using the merge sort algorithm**

**Objectives:**

* To implement the **Merge Sort algorithm** using recursion.
* To divide an array into two halves, sort them, and merge them in sorted order.
* To demonstrate the working of **Merge Sort** on an example array.

**Theory:**

**Merge Sort** is a **divide and conquer** algorithm that works as follows:

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

**Properties of Merge Sort:**

* **Stable Sorting Algorithm:** Maintains relative order of equal elements.
* **Not In-Place:** Requires extra space for merging.
* **Used in Large Datasets:** Efficient for sorting large data structures.

**Time and Space Complexity:**

* **Time Complexity:**
  + **Best Case:** O(nlogn)
  + **Worst Case:** O(nlogn)
  + **Average Case:**O(nlogn)
* **Space Complexity:** O(n) (Due to extra arrays used in merging)

**Algorithm (Step by Step):**

1. **Base Case:** If l >= r, return (array is already sorted).
2. **Divide:** Find the middle index mid = (l + r) / 2.
3. **Recursively Sort:**
   * Sort left half (mergesort(arr, l, mid)).
   * Sort right half (mergesort(arr, mid+1, r)).
4. **Merge:** Combine the sorted halves using merge() function.
5. **Print the sorted array.**

Code :

#include <bits/stdc++.h> // Include all standard libraries

using namespace std;

// Function to merge two sorted halves

void merge(int arr[], int l, int mid, int r) {

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

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

int a[n1], b[n2]; // Temporary arrays to hold values

// Copy elements into left subarray

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

a[i] = arr[l + i];

}

// Copy elements into right subarray

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

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

}

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

// Merge the two arrays back into arr[]

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

if (a[i] < b[j]) {

arr[k] = a[i]; // Place smaller element first

i++;

} else {

arr[k] = b[j];

j++;

}

k++;

}

// Copy remaining elements of a[], if any

while (i < n1) {

arr[k] = a[i];

i++;

k++;

}

// Copy remaining elements of b[], if any

while (j < n2) {

arr[k] = b[j];

j++;

k++;

}

}

// Recursive function to apply merge sort

void mergesort(int arr[], int l, int r) {

if (l < r) {

int mid = (l + r) / 2;

// Recursively sort first and second halves

mergesort(arr, l, mid);

mergesort(arr, mid + 1, r);

// Merge the sorted halves

merge(arr, l, mid, r);

}

}

// Main function

int main() {

int arr[] = {5, 4, 3, 2, 1}; // Input array

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

mergesort(arr, 0, n - 1); // Call merge sort

// Print the sorted array

cout << "Sorted Array: ";

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

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

}

cout << endl;

return 0;

}

**Input and Output:**

Input Array: 5 4 3 2 1

Sorted Array: 1 2 3 4 5

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

**Objectives:**

* To implement the **Binary Search algorithm** for finding an element in a sorted array.
* To return the index of the element if found, or print "No" if the element is not present.

**Theory:**

**Binary Search** is a highly efficient algorithm for finding an element in a **sorted array**:

* It works by repeatedly dividing the search interval in half.
* If the value of the search key is less than the item in the middle of the interval, the search continues on the left half, or on the right half if the value is greater.
* **Time Complexity:** O(logn) (much faster than linear search for large arrays).

**Steps in Binary Search:**

1. Start with two pointers, l (left) and r (right), pointing to the first and last index of the array.
2. Find the middle index mid = (l + r) / 2.
3. If the element at mid is the target, return the index.
4. If the target is greater than the element at mid, search the right half (l = mid + 1).
5. If the target is smaller, search the left half (r = mid - 1).
6. If the element is not found, return "No".

**Time and Space Complexity:**

* **Time Complexity:** O(logn) (because we reduce the search space by half in each iteration).
* **Space Complexity:** O(1) (since we use a constant amount of extra space).

**Algorithm (Step by Step):**

1. Read n (size of the array) and a[] (array of integers).
2. Read x (the element to search for).
3. Initialize l = 0 and r = n - 1, and index = -1.
4. While l <= r:
   * Calculate the middle index mid = (l + r) / 2.
   * If a[mid] == x, store mid in index and exit the loop.
   * If x > a[mid], set l = mid + 1 to search the right half.
   * If x < a[mid], set r = mid - 1 to search the left half.
5. After the loop, check if index is -1. If true, print "No". Otherwise, print Index: index.

**Code:**

#include<bits/stdc++.h>

using namespace std;

int main()

{

   int n;

   cin>>n;

   int a[n];

   for(int i=0;i<n;i++)

   {

    cin>>a[i];

   }

   int x;

   cin>>x;

   int l=0;

   int r=n-1;

   int index=-1;

while(l<=r)

   {

      int mid=(l+r)/2;

      if(a[mid]==x)

      {

         index=mid;

         break;

      }

      else if(x>a[mid])

      {

         l=mid+1;

      }

      else

      {

         r=mid-1;

      }

   }

 if(index==-1)

   {

      cout<<"No";

   }

   else

   cout<<"Index :"<<index;

   return 0;

}

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

**Objectives:**

* To implement the **Naive Pattern Matching Algorithm** to find the occurrence of a substring (pattern) within a string (text).
* To return the index where the pattern starts in the text if it is found, or indicate that the pattern was not found.

**Theory:**

The **Naive Pattern Matching Algorithm** works by:

1. Scanning the text from the beginning to the end.
2. At each position, comparing the substring of length m (the pattern length) with the pattern.
3. If a match is found, the index is returned.
4. If no match is found after scanning the entire text, -1 is returned.

**Time and Space Complexity:**

* **Time Complexity:**
  + **Best Case:** O(n) (when the pattern is found early in the text).
  + **Worst Case:** O(n⋅m) (when the pattern is not found, or every comparison must be checked).
  + **Average Case:**O(n⋅m).
* **Space Complexity:**
  + **Auxiliary Space:** O(1), as we are using only a constant amount of extra space, excluding the input and output.

**Algorithm (Step by Step):**

1. Read the text and pattern.
2. Calculate the length of the text (n) and pattern (m).
3. Iterate through the text from index 0 to n - m:
   * For each index, check if the substring of length m starting at index i in text is equal to pattern.
   * If a match is found, return the index i.
4. If the loop completes without finding a match, return -1.
5. In the main() function, output the result based on the returned index.

**Code**

#include <iostream>

#include <string>

using namespace std;

// Function to perform pattern matching

int patternMatching(string text, string pattern) {

int n = text.length(); // Length of the text

int m = pattern.length(); // Length of the pattern

// Loop through the text to find the pattern

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

// Compare the substring of length m from text with the pattern

if (text.substr(i, m) == pattern) {

return i; // Pattern found, return the starting index

}

}

return -1; // Pattern not found

}

int main() {

string text, pattern;

// Input text and pattern from user

cout << "Enter the text: ";

cin >> text;

cout << "Enter the pattern: ";

cin >> pattern;

// Perform pattern matching

int result = patternMatching(text, pattern);

// Output the result

if (result == -1) {

cout << "Pattern not found!";

} else {

cout << "Pattern found at index: " << result;

}

return 0;

}

Input:

Enter the text: hello

Enter the pattern: ll

Output: 3

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

**Objectives:**

* To implement a **custom queue** using C++ that mimics the functionality of a queue.
* To perform operations such as **enqueue**, **dequeue**, **peek**, **deleteElement**, **display**, and **size** on the queue.

**Theory:**

A **Queue** is a linear data structure that follows the **First In First Out (FIFO)** principle:

* **Enqueue**: Adds an element to the end of the queue.
* **Dequeue**: Removes an element from the front of the queue.
* **Peek**: Returns the element at the front without removing it.
* **Delete Element**: Similar to dequeue, removes the front element.
* **Display**: Shows all elements in the queue.
* **Size**: Returns the number of elements in the queue.

In this implementation:

* A **queue** from C++ Standard Library (queue<int> v) is used to store the elements.
* A custom class myqueue is created to encapsulate the queue and its associated operations.

**Time and Space Complexity:**

* **Time Complexity:**
  + **Enqueue**: O(1) (inserting at the back of the queue).
  + **Dequeue**: O(1) (removing from the front of the queue).
  + **Peek**: O(1) (accessing the front element).
  + **DeleteElement**: O(1) (removing from the front of the queue).
  + **Display**:)O(n) (traversing all elements in the queue).
  + **Size**: O(1) (fetching the size of the queue).
* **Space Complexity:**
  + **Auxiliary Space:** O(n) (for storing the elements in the queue).

**Algorithm (Step by Step):**

1. **Create a class myqueue** that encapsulates a queue<int>:
   * Implement functions for **enqueue**, **dequeue**, **peek**, **deleteElement**, **display**, and **size**.
2. **In the main() function**:
   * Create an object of myqueue.
   * Ask for the number of elements to enqueue and enqueue them.
   * Perform operations like **dequeue**, **peek**, **deleteElement**, and **display** the queue.

**Code:**

#include <bits/stdc++.h>

using namespace std;

// Class definition for custom queue

class myqueue

{

public:

queue<int> v; // Use the standard queue to store elements

// Function to add an element to the queue

void enqueue(int val)

{

v.push(val); // Push element to the queue

cout << "Enqueued: " << val << endl;

}

// Function to remove the front element from the queue

void dequeue()

{

if (!v.empty()) // Check if queue is not empty

{

cout << "Dequeued: " << v.front() << endl;

v.pop(); // Pop the front element

}

else

{

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

}

}

// Function to get the front element without removing it

void peek()

{

if (!v.empty()) // Check if queue is not empty

{

cout << "Front item: " << v.front() << endl;

}

else

{

cout << "Queue is empty. No front item to peek" << endl;

}

}

// Function to delete the front element (Similar to dequeue)

void deleteElement()

{

if (!v.empty()) // Check if queue is not empty

{

cout << "Deleted: " << v.front() << endl;

v.pop(); // Pop the front element

}

else

{

cout << "Queue is empty. Cannot delete" << endl;

}

}

// Function to display all elements in the queue

void display()

{

if (v.empty()) // Check if queue is empty

{

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

}

else

{

queue<int> temp = v; // Temporary queue to display elements

cout << "Queue elements: ";

while (!temp.empty()) // Traverse and print all elements

{

cout << temp.front() << " ";

temp.pop();

}

cout << endl;

}

}

// Function to display the size of the queue

void Size()

{

cout << "Size: " << v.size() << endl; // Output the size of the queue

}

};

// Main function

int main()

{

myqueue q; // Create a queue object

int n;

cout << "Enter number of elements to enqueue: ";

cin >> n; // Input number of elements

// Loop to enqueue elements

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

{

int x;

cout << "Enter element " << i + 1 << ": ";

cin >> x; // Input each element

q.enqueue(x); // Enqueue the element

q.display(); // Display queue after each enqueue

}

q.Size(); // Display size of the queue

q.dequeue(); // Perform dequeue operation

q.peek(); // Peek the front element

q.deleteElement(); // Delete the front element

q.display(); // Display the queue after operations

return 0;

}

**Input and output:**

Enter number of elements to enqueue: 3

Enter element 1: 10

Enqueued: 10

Queue elements: 10

Enter element 2: 20

Enqueued: 20

Queue elements: 10 20

Enter element 3: 30

Enqueued: 30

Queue elements: 10 20 30

Size: 3

Dequeued: 10

Front item: 20

Deleted: 20

Queue elements: 30

**Lab-7: Write a program to solve the n-queens problem using backtracking**

**Objectives:**

* To solve the N-Queens problem by placing N queens on an NxN board such that no two queens attack each other.
* Use **backtracking** to explore possible placements and reject invalid configurations.

**Theory:**

1. **Backtracking Approach**:
   * Start placing queens row by row.
   * For each row, try placing a queen in each column.
   * Check if the queen placement is safe by ensuring it doesn't conflict with queens already placed.
   * If placement is safe, move to the next row.
   * If placing a queen leads to a valid solution, print it; otherwise, backtrack and try the next column for the current row.
2. **Safe Check**:
   * The function isSafe() ensures that no two queens are attacking each other by checking:
     + Column conflicts: No queen should share the same column.
     + Diagonal conflicts: No queen should be on the same diagonal (both primary and secondary diagonals).

**Time and Space Complexity:**

* **Time Complexity:**
  + In the worst case, we need to check all possible placements of queens. The number of possible placements is N! because for each row, there are N possible columns for placing the queen.
  + Therefore, the time complexity is O(N!).
* **Space Complexity:**
  + The space complexity is O(N) because we store the current configuration of queens in an array of size N (placed[]).

**Algorithm (Step by Step):**

1. **Initialization**:
   * Start by initializing an array placed[] of size N, where each element represents the column of the queen in that row.
2. **Backtracking Function (solveNQueens)**:
   * If row == N, print the solution (i.e., all queens are placed).
   * Otherwise, try placing a queen in each column of the current row and check if it is safe.
   * If safe, recursively place queens in the next row.
3. **Safety Check (isSafe)**:
   * For each row, check if placing a queen at a given column causes a conflict with any previously placed queens:
     + No other queen should be in the same column.
     + No other queen should be on the same diagonal.
4. **Output**:
   * Print all valid solutions, where each solution is a placement of queens such that no two queens threaten each other.

**Code with Explanations:**

#include <iostream>

using namespace std;

#define N 4 // Define the size of the board (4x4 for this case)

// Function to print the board configuration

void printSolution(int placed[]) {

static int solutionCount = 0; // Keeps track of the solution number

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

// Loop through each row and column to print the chessboard

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

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

// Print 'Q' if the queen is placed in this position, otherwise print '.'

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

}

// Function to check if it's safe to place a queen at (row, col)

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

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

// Check if any queen is in the same column or diagonals

if (placed[prev] == col || // Same column

placed[prev] - prev == col - row || // Same diagonal (\)

placed[prev] + prev == col + row) { // Same diagonal (/)

return false; // Not safe to place a queen here

}

}

return true; // Safe to place a queen here

}

// Recursive function to solve N-Queens problem

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

if (row == N) { // All queens are placed, print the solution

printSolution(placed);

return;

}

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

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

if (isSafe(placed, row, col)) { // Check if it's safe

placed[row] = col; // Place the queen

solveNQueens(placed, row + 1); // Recur for the next row

}

}

}

// Main function

int main() {

int placed[N] = {-1}; // Initialize an array to keep track of queen positions

solveNQueens(placed, 0); // Start solving from the first row

return 0;

}

**Input:**

N=4

**Output:**

Solution 1:

. Q . .

. . . Q

Q . . .

. . Q .

Solution 2:

. . Q .

Q . . .

. . . Q

. Q . .

**Lab 8: Consider a set S={5,10,12,13,15,18} and d=30. Write a program to solve the sum of subsets problem.**

**Objectives:**

* To find all subsets of an array whose sum equals the target value.
* Use **backtracking** to explore all possible combinations of array elements and check their sums.

**Theory:**

1. **Backtracking Approach**:
   * The function findSubsets tries to find subsets by considering each element of the array.
   * For each element, we have two options:
     + **Include the element** in the subset and check if the sum equals the target.
     + **Exclude the element** from the subset and continue searching.
2. **Stopping Condition**:
   * The function stops exploring a subset when:
     + The sum of the subset equals the target, and the subset is printed.
     + The index exceeds the array bounds, or the sum exceeds the target.
3. **Recursive Exploration**:
   * At each step, we either include the current element in the subset or exclude it and explore both possibilities recursively.

**Time and Space Complexity:**

* **Time Complexity**:
  + Since we are exploring all subsets of the array, the number of subsets is 2n (where n is the number of elements in the array).
  + The time complexity is O(2n) because we check each possible subset.
* **Space Complexity**:
  + The space complexity is O(n) because of the recursive function call stack, where n is the maximum depth of the recursion.

**Algorithm (Step by Step):**

1. **Initialization**:
   * Read the input array arr[] and the target sum.
2. **Recursive Function findSubsets**:
   * If the current subset sum equals the target, print the subset.
   * If the current index exceeds the array size or the sum exceeds the target, stop exploring that path.
   * Otherwise, explore two options:
     + Include the current element and move to the next index.
     + Exclude the current element and move to the next index.
3. **Output**:
   * Print all valid subsets whose sum equals the target.

**Code :**

#include <iostream>

using namespace std;

// Function to find all subsets whose sum equals the target

void findSubsets(int arr[], int n, int index, int sum, int target, string current) {

// If the current sum equals the target, print the subset

if (sum == target) {

cout << "{ " << current << "}" << endl;

return;

}

// If index reaches the end of the array or the sum exceeds target, stop

if (index == n || sum > target) {

return;

}

// Include the current element in the subset and recur

findSubsets(arr, n, index + 1, sum + arr[index], target, current + to\_string(arr[index]) + " ");

// Exclude the current element from the subset and recur

findSubsets(arr, n, index + 1, sum, target, current);

}

int main() {

int n, target;

// Input the number of elements and target sum

cout << "Enter number of elements: ";

cin >> n;

int arr[n];

cout << "Enter elements: ";

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

cin >> arr[i];

}

cout << "Enter target sum: ";

cin >> target;

cout << "Subsets with sum " << target << " are:" << endl;

findSubsets(arr, n, 0, 0, target, "");

return 0;

}

Input:

Enter number of elements: 4

Enter elements: 1 2 3 4

Enter target sum: 5

Output:

Subsets with sum 5 are:

{ 1 4 }

{ 2 3 }

{ 1 2 3 }

**Lab-9: Write a program to solve the 0/1 Knapsack problem using a dynamic programming approach with profits P=(15,25,13,23) weights W=(2,6,12,9) knapsack capacity C=20, and the number of items n=4**

**Objectives:**

* To maximize the profit by selecting items such that the total weight does not exceed the given knapsack capacity.
* Solve using dynamic programming to efficiently compute the optimal solution.

**Theory:**

* The problem is solved using a bottom-up approach, where we use a dynamic programming (DP) table to store the maximum profit achievable for each item and weight combination.
* The knapsack can either include or exclude an item based on whether its weight fits into the remaining capacity.

**Time Complexity:**

* **Time Complexity**: O(n \* capacity), where n is the number of items and capacity is the total knapsack capacity.
* This is because we are iterating through all items and all possible weights (up to the knapsack capacity).

**Space Complexity:**

* **Space Complexity**: O(n \* capacity), due to the space needed to store the DP table.

**Algorithm (Step-by-step):**

1. **Input**: Number of items n, their respective profits, and weights, and the knapsack's capacity.
2. Initialize a DP table k[][] of size (n+1) x (capacity+1) with all values set to 0.
3. **Filling the DP Table**:
   * Loop through each item (i) and weight (w):
     + If the current item's weight is less than or equal to w, find the maximum profit between:
       - Excluding the item (take the value from the row above).
       - Including the item (add its profit to the maximum value obtained by subtracting its weight from the current capacity).
     + If the item’s weight is greater than w, carry forward the previous value.
4. The value at k[n][capacity] will hold the maximum achievable profit.

**Code :**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n, capacity;

// Taking input from the user for the number of items

cout << "Enter number of items: ";

cin >> n;

int profit[n], weight[n];

// Taking input for the profit of each item

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

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

{

cin >> profit[i];

}

// Taking input for the weight of each item

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

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

{

cin >> weight[i];

}

// Taking input for the knapsack's total capacity

cout << "Enter knapsack capacity: ";

cin >> capacity;

// Creating the DP table (initializing all values to 0)

int k[n + 1][capacity + 1] = {0};

// Filling the DP table using the 0/1 Knapsack approach

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

{

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

{

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

{

k[i][w] = 0; // First row and column will always be 0

}

else if (weight[i - 1] <= w)

{

// If the item can fit in the knapsack, choose the maximum of including or excluding it

k[i][w] = max(k[i - 1][w], profit[i - 1] + k[i - 1][w - weight[i - 1]]);

}

else

{

// If the item cannot fit, copy the value from the previous row

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

}

}

}

// Printing the maximum profit that can be achieved with the given knapsack capacity

cout << "Maximum Profit: " << k[n][capacity] << endl;

return 0;

}

**Input**:

Enter number of items: 4

Enter the profits of items: 10 40 30 50

Enter the weights of items: 5 4 6 3

Enter knapsack capacity: 10

**Output:**

Maximum Profit: 90

**Lab-10: Write a program to solve the Tower of Hanoi problem for N disks.**

**Objectives:**

* The objective of the Tower of Hanoi problem is to move all the disks from a source rod to a destination rod, following the following 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.

**Theory:**

* The problem is solved recursively:
  + Move the top n-1 disks from the source to the auxiliary rod.
  + Move the nth (largest) disk from the source to the destination rod.
  + Move the n-1 disks from the auxiliary rod to the destination rod.

**Time Complexity:**

* **Time Complexity**: O(2n), where n is the number of disks. This is because the number of moves required grows exponentially with the number of disks.

**Space Complexity:**

* **Space Complexity**: O(n), since we are using recursion and the maximum recursion depth is n.

**Algorithm (Step-by-step):**

1. If there is only one disk, move it directly from the source rod to the destination rod.
2. Otherwise, move n-1 disks from the source rod to the auxiliary rod using the destination rod.
3. Move the nth disk (largest) directly from the source rod to the destination rod.
4. Finally, move the n-1 disks from the auxiliary rod to the destination rod using the source rod.

**Code Explanation:**

#include<bits/stdc++.h>

using namespace std;

// Recursive function for Tower of Hanoi

void Toh(int n, char src, char aux, char dest)

{

// Base case: If only one disk is left, move it directly

if(n == 1) {

cout << "Move disk- " << n << " from " << src << " To " << dest << endl;

return;

}

// Move the top n-1 disks from source to auxiliary rod

Toh(n-1, src, dest, aux);

// Move the nth disk (largest) from source to destination rod

cout << "Move disk - " << n << " from " << src << " To " << dest << endl;

// Move the n-1 disks from auxiliary rod to destination rod

Toh(n-1, aux, src, dest);

}

int main()

{

int n;

cin >> n; // Input number of disks

Toh(n, 'A', 'B', 'C'); // Call the function with source 'A', auxiliary 'B', and destination 'C'

return 0;

}

Input:

N=3

Output:

Move disk- 1 from A To C

Move disk - 2 from A To B

Move disk- 1 from C To B

Move disk - 3 from A To C

Move disk- 1 from B To A

Move disk - 2 from B To C

Move disk- 1 from A To C