**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**



**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

**Course name: Data Structure and Algorithm sessional**

**Course Code: ICE-2202**

Submitted By:

**Name: Md. Uzzal Mia**

**Roll:** 220605

**Reg:** 1065452

**Session:**2021-22

2nd Year 2nd Semester

Dept. of ICE, PUST

Submitted To:

**Md. Anwar Hossain**

Professor

Department of Information and Communication Engineering

Pabna University of Science and Technology

Pabna, Bangladesh

**Index**

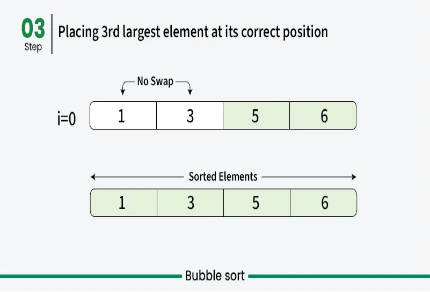
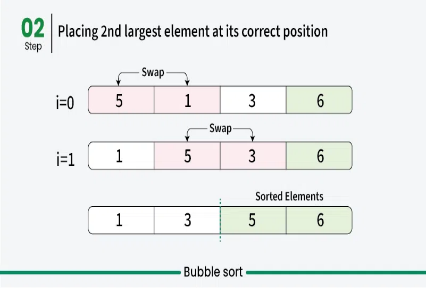
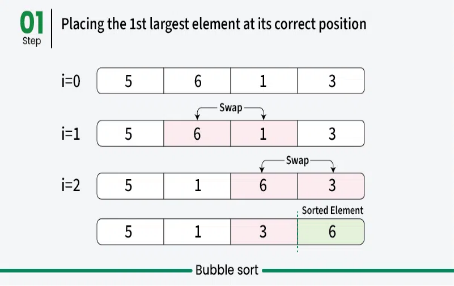
|  |  |
| --- | --- |
| Problem Number | Title |
| 01 | Write a program to sort a linear array using the bubble sort algorithm. |
| 02 | Write a program to find an element using a linear search algorithm. |
| 03 | Write a program to sort a linear array using the merge sort algorithm. |
| 04 | Write a program to find an element using the binary search algorithm |
| 05 | Write a program to find a given pattern from text using the pattern matching algorithm. |
| 06 | Write a program to implement a queue data structure along with its typical operations. |
| 07 | Write a program to solve n queen's problem using backtracking. |
| 08 | Consider a set S= (5, 10, 12, 13, 15, 18) and d = 30 Write a program to solve the sum of subset problem. |
| 09 | 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 number: 1**

**Title: Write a program to sort a linear array using the bubble sot algorithm**

**Logic**: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

* We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
* In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
* In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.



**Bubble Sort Algorithm**

**Step 1** − Check if the first element in the input array is greater than the next element in the array.

**Step 2** − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

**Step 3** − Repeat Step 2 until we reach the end of the array.

**Step 4** − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

**Step 5** − The final output achieved is the sorted array

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

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

    cin >> n;

    int a[n];

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

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

    {

        cin >> a[i];

    }

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

    {

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

        {

*if* (a[j] > a[j + 1])

            {

                swap(a[j], a[j + 1]);

            }

        }

    }

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

    {

        cout << a[i] << " ";

    }

*return* 0;

}

**Input:**

Enter the size of Array: 8

Enter elements of the array: 32 51 27 85 66 23 13 57

**Output:**

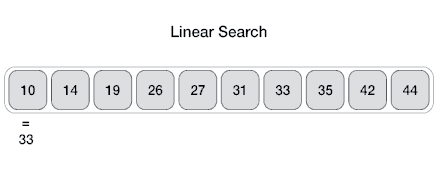
Sorted array: 13 23 27 32 51 57 66 85

**Problem Number: 2**

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

**Logic:** Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

For instance, in the given animated diagram, we are searching for an element 33. Therefore, the linear search method searches for it sequentially from the very first element until it finds a match. This returns a successful search.



In the same diagram, if we have to search for an element 46, then it returns an unsuccessful search since 46 is not present in the input

**Linear Search Algorithm**

**Step 1** − Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

**Step 2** − If the value matches with the key, return the position at which the value was found.

**Step 3** − If the value does not match with the key, compare the next element in the array.

**Step 4** − Repeat Step 3 until there is a match found. Return the position at which the match was found.

**Step 5** − If it is an unsuccessful search, print that the element is not present in the array and exit the program.

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

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

    cin >> n;

    int arr[n];

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

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

    {

        cin >> arr[i];

    }

    int target;

    cout << "Enter the element you want to find: ";

    cin >> target;

    int result = 0;// *Flag variable to indicate if the element is found*

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

    {

*if* (arr[i] == target)

        {

            cout << "Found at position " << i << endl;

            result = 1;// *Update the flag if the element is found*

*break*;

        }

    }

*if* (result == 0)// *Check if the flag was updated*

    {

        cout << "Element not found." << endl;

    }

*return* 0;

}

**Input:**

Enter size of the linear array: 6

Enter elements of the array: 12 48 44 32 10 50

Enter the element you want to find: 32

**Output**

Found at position 3

**Problem Number: 3**

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

**Logic:** The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

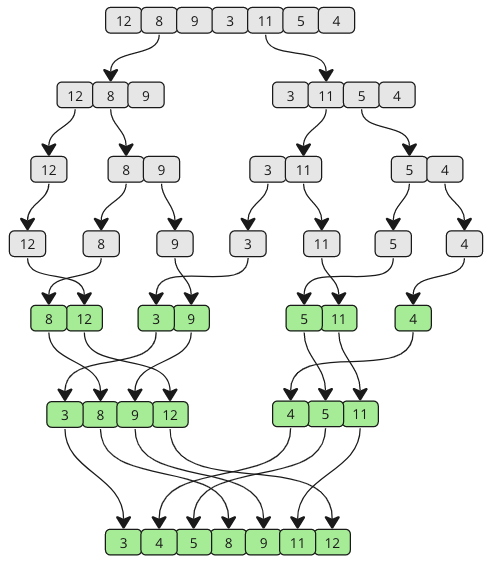
**Divide:**The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

**Conquer:**The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

The breaking down and building up of the array to sort the array is done recursively.

In the animation above, each time the bars are pushed down represents a recursive call, splitting the array into smaller pieces. When the bars are lifted up, it means that two sub-arrays have been merged together.

Take a look at the drawing below to see how Merge Sort works from a different perspective. As you can see, the array is split into smaller and smaller pieces until it is merged back together. And as the merging happens, values from each sub-array are compared so that the lowest value comes first.



**Merge Sort Algorithm**

**Step 1 -** Divide the unsorted array into two sub-arrays, half the size of the original.

**Step 2 -** Continue to divide the sub-arrays as long as the current piece of the array has more than one element.

**Step 3 -** Merge two sub-arrays together by always putting the lowest value first.

**Step 4 -** Keep merging until there are no sub-arrays left.

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

void merge(int a[], int l, int mid, int r)

{

    int leftSize = mid - l + 1;

    int rightSize = r - mid;

    int L[leftSize], R[rightSize];

    int k = 0;

*for* (int i = l; i <= mid; i++)

    {

        L[k] = a[i];

        k++;

    }

    k = 0;

*for* (int i = mid + 1; i <= r; i++)

    {

        R[k] = a[i];

        k++;

    }

    int i = 0, j = 0;

    int cur = l;

*while* (i < leftSize && j < rightSize)

    {

*if* (L[i] <= R[j])

        {

            a[cur] = L[i];

            i++;

        }

*else*

        {

            a[cur] = R[j];

            j++;

        }

        cur++;

    }

*while* (i < leftSize)

    {

        a[cur] = L[i];

        i++;

        cur++;

    }

*while* (j < rightSize)

    {

        a[cur] = R[j];

        j++;

        cur++;

    }

}

void merge\_sort(int a[], int l, int r)

{

*if* (l < r)

    {

        int mid = (l + r) / 2;

        merge\_sort(a, l, mid);

        merge\_sort(a, mid + 1, r);

        merge(a, l, mid, r);

    }

}

int main()

{

    int n;

    cin >> n;

    int a[n];

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

    {

        cin >> a[i];

    }

    merge\_sort(a, 0, n - 1);

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

    {

        cout << a[i] << " ";

    }

*return* 0;

}

**Input:**

Enter size of the array: 7

Enter elements of the array: 12 8 9 3 11 5 4

**Output:**

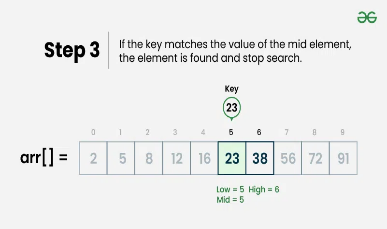
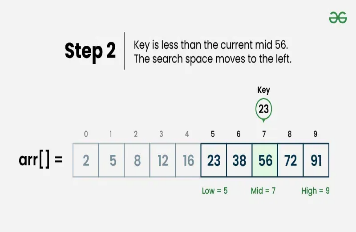
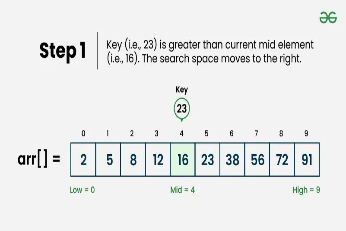
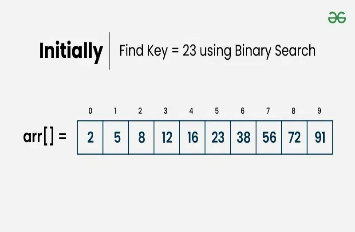
Merge sorted array: 3 4 5 8 9 11 12

**Problem Number: 4**

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

**Logic:** Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.



**Binary Search Algorithm**

**Step 1** − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

**Step 2** − If it does not match the key value, check if the key value is either greater than or less than the median value.

**Step 3** − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

**Step 4** − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

**Step 5** − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

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

    cin >> n;

    int a[n];

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

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

    {

        cin >> a[i];

    }

    sort(a, a + n);

    int target;

    cout << "Enter the element you want to find: ";

    cin >> target;

    int l = 0;

    int r = n - 1;

    bool result = false;

*while* (l <= r)

    {

        int mid = (l + r) / 2;

*if* (a[mid] == target)

        {

            cout << "Find the position " << mid << endl;

            result = true;

*break*;

        }

*else* *if* (a[mid] < target)

        {

            l = mid + 1;

        }

*else*

        {

            r = mid - 1;

        }

    }

*if* (result == 0)

    {

        cout << "target value not found" << endl;

    }

*return* 0;

}

**Input:**

Enter the size of Array: 10

Enter elements of the array: 2 5 8 12 16 23 38 56 72 91

Enter the element you want to find: 23

**Output:**

Find the position 5

**Problem Number: 5**

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

**Logic:** Naive pattern searching is the simplest method among other pattern searching algorithms. Although, it is more efficient than the brute force approach, however, it is not the most optimal method available. Like brute force, it also checks for all characters of the main string in order to find the pattern. Hence, its time complexity is O(m\*n) where the 'm' is the size of pattern and 'n' is the size of the main string. This algorithm is helpful for smaller texts only.

The naive pattern searching algorithm does not require any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation. If a match is found, the end result of the pattern matching operation will be the index of specified pattern, otherwise -1. Furthermore, this operation can return all the indices if the desired pattern appears multiple times within the main string.

Let's understand the input-output scenario of a pattern matching problem with an example –

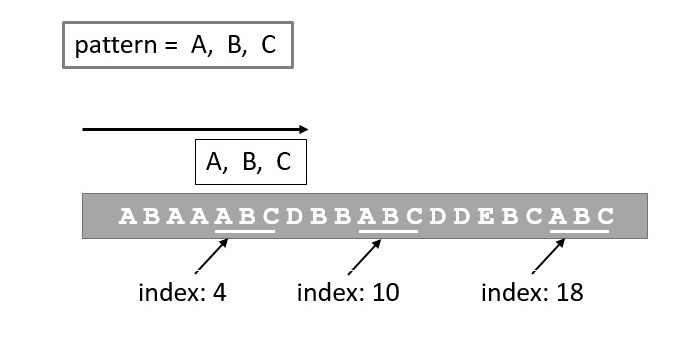


Figure: Pattern Matching

**Algorithm**

**Step 1 -** Start

**Step 2** - Input text and pattern.

**Step 3 -** Calculate the length of text as textLen.

**Step 4 -** Calculate the length of pattern as patternLen.

**Step 5 -** Initialize a Boolean variable found as false.

**Step 6 -** Loop for i from 0 to textLen - patternLen:

Initialize j to 0.

Inner Loop for j from 0 to patternLen - 1:

Compare text[i + j] with pattern[j].

If they are not equal, break the inner loop.

If j equals patternLen (all characters matched):

Print "Pattern found at position: i".

Set found to true.

**Step 7 -** End Loop

**Step 8 -** If found is still false:

Print "Pattern not found in the text".

**Step 9 -** End

**Source Code**

#*include* <iostream>

using namespace std;

int main() {

    string text, pattern;

// *Input the text*

    cout << "Enter the text: ";

    getline(cin, text);

// *Input the pattern to search for*

    cout << "Enter the pattern: ";

    getline(cin, pattern);

    int textLen = text.length();

    int patternLen = pattern.length();

    bool found = false;

// *Loop through the text up to the point where the pattern can fit*

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

        int j;

// *Check the current substring with the pattern*

*for* (j = 0; j < patternLen; j++) {

*if* (text[i + j] != pattern[j]) {

*break*;

            }

        }

*if* (j == patternLen) {

            cout << "Pattern found at position: " << i << endl;

            found = true;// *Set found to true*

        }

    }

// *If no match is found*

*if* (!found) {

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

    }

*return* 0;

}

**Input:**

Enter the text: ABAAABCDBBABCDDEBCABC

Enter the pattern: ABC

**Output:**

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18

**Problem Number: 6**

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

**Logic:** Think of a queue as people standing in line in a supermarket.

The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO: First In First Out.

Basic operations we can do on a queue are

* **Enqueue:**Adds a new element to the queue.
* **Dequeue:**Removes and returns the first (front) element from the queue.
* **Peek:**Returns the first element in the queue.
* **isEmpty:**Checks if the queue is empty.
* **Size:**Finds the number of elements in the queue.

**Algorithm**

**Step 1 -** START

**Step 2 -** Check if the queue is full.

**Step 3 -** If the queue is full, produce overflow error and exit.

**Step 4 -** If the queue is not full, increment rear pointer to point the next empty space.

**Step 5 -** Add data element to the queue location, where the rear is pointing.

**Step 6 -** return success.

**Step 7 -** END

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

// *Class definition for custom queue*

*class* myqueue

{

public:

    queue<int> v;

// *Function to add an element to the queue*

    void enqueue(int val)

    {

        v.push(val);

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

    }

// *Function to remove the front element from the queue*

    void dequeue()

    {

*if* (!v.empty())

        {

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

            v.pop();

        }

*else*

        {

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

        }

    }

// *Function to get the front element without removing it*

    void peek()

    {

*if* (!v.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())

        {

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

            v.pop();

        }

*else*

        {

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

        }

    }

// *Function to display all elements in the queue*

    void display()

    {

*if* (v.empty())

        {

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

        }

*else* // *If queue has elements*

        {

            queue<int> temp = v;

            cout << "Queue elements: ";

*while* (!temp.empty())

            {

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

                temp.pop();

            }

            cout << endl;

        }

    }

// *Function to display the size of the queue*

    void Size()

    {

        cout << "Size: " << v.size() << endl;

    }

};

// *Main function*

int main()

{

    myqueue q;

    int n;

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

    cin >> n;

// *Loop to enqueue elements*

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

    {

        int x;

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

        cin >> x;

        q.enqueue(x);

        q.display();

    }

    q.Size();

    q.dequeue();

    q.peek();

    q.deleteElement();

    q.display();

*return* 0;

}

**Input:**

Enter number of elements to enqueue: 5

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

Enter element 4: 40

Enqueued: 40

Queue elements: 10 20 30 40

Enter element 5: 50

Enqueued: 50

**Output:**

Queue elements: 10 20 30 40 50

Size: 5

Dequeued: 10

Front item: 20

Deleted: 20

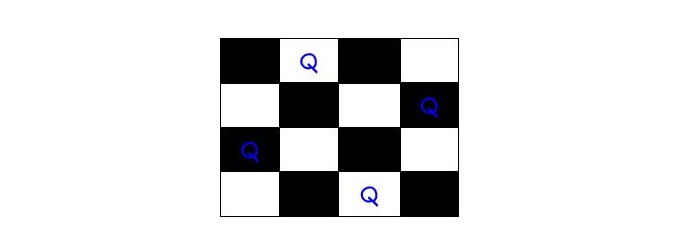
Queue elements: 30 40 50

**Problem Number: 7**

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

**Logic**: In N-Queen problem, we are given an N x N chessboard and we have to place N number of queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. The most popular approach for solving the N Queen puzzle is Backtracking.

suppose the given chessboard is of size 4x4 and we have to arrange exactly 4 queens in it. The solution arrangement is shown in the figure below



In the naive method to solve n queen problem, the algorithm generates all possible solutions. Then, it explores all of the solutions one by one. If a generated solution satisfies the constraint of the problem, it prints that solution.

**N Queens Problem Algorithm**

Step 1 - Place the first queen in the top-left cell of the chessboard.

**Step 2 -** After placing a queen in the first cell, mark the position as a part of the solution and then recursively check if this will lead to a solution.

**Step 3 -** Now, if placing the queen doesn’t lead to a solution. Then go to the first step and place queens in other cells. Repeat until all cells are tried.

**Step 4 -** If placing queen returns a lead to solution return TRUE.

**Step 5 -** If all queens are placed return TRUE.

**Step 6 -** If all rows are tried and no solution is found, return FALSE.

**Source Code**

#*include* <iostream>

using namespace std;

int solutionCount = 0;

void printSolution(int placed[], int N) {

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

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

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

*if* (placed[i] == j)

                cout << "Q ";

*else*

                cout << ". ";

        }

        cout << endl;

    }

}

// *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++) {

*if* (placed[prev] == col ||

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

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

*return* false;

        }

    }

*return* true;

}

// *Function to solve N-Queens using recursion and backtracking*

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

*if* (row == N) {

        printSolution(placed, N);

*return*;

    }

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

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

            placed[row] = col;// *Place queen at (row, col)*

            solveNQueens(placed, row + 1, N);

        }

    }

}

int main() {

    int N;

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

    cin >> N;

    int placed[N];

    fill\_n(placed, N, -1);

    solveNQueens(placed, 0, N);

*if* (solutionCount == 0)

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

*return* 0;

}

**Input:**

Enter the number of queens (N): 4

**Output:**

Solution 1:

. Q . .

. . . Q

Q . . .

. . Q .

Solution 2:

. . Q .

Q . . .

. . . Q

. Q . .

**Problem Number: 8**

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

**Logic:** In the sum of subsets problem, there is a given set with some non-negative integer elements. And another sum value is also provided, our task is to find all possible subsets of the given set whose sum is the same as the given sum value.

*Set: In mathematical terms, a set is defined as a collection of similar types of objects. The entities or objects of a set must be related to each other through the same rule.*

*Subset: Suppose there are two sets namely set P and set Q. The set P is said to be a subset of set Q, only if all the elements of set P also belong to the set Q and vice-versa need not be true.*

Input Output Scenario

Suppose the given set and sum value is −

Set = {1, 9, 7, 5, 18, 12, 20, 15}

sum value = 35

All possible subsets of the given set, where sum of each element for every subset is the same as the given sum value are given below −

{1 9 7 18}

{1 9 5 20}

{5 18 12}

In the naive method to solve a subset sum problem, the algorithm generates all the possible permutations and then checks for a valid solution one by one. Whenever a solution satisfies the constraints, mark it as a part of the solution.

In solving the subset sum problem, the backtracking approach is used for selecting a valid subset. When an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.

In the worst-case scenario, backtracking approach may generate all combinations, however, in general, it performs better than the naive approach.

**Algorithm**

**Step 1 -** First, take an empty subset.

**Step 2 -** Include the next element, which is at index 0 to the empty set.

**Step 3** - If the subset is equal to the sum value, mark it as a part of the solution.

**Step 4 -** If the subset is not a solution and it is less than the sum value, add next element to the subset until a valid solution is found.

**Step 5 -** Now, move to the next element in the set and check for another solution until all combinations have been tried

**Source Code**

#*include* <iostream>

using namespace std;

int main()

{

    int N, targetSum;

// *Taking user input*

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

    cin >> N;

    int S[N];

    cout << "Enter the elements: ";

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

    {

        cin >> S[i];// *Input array elements*

    }

    cout << "Enter the target sum: ";

    cin >> targetSum;

    int totalSubsets = 1;

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

    {

        totalSubsets \*= 2;// *Equivalent to 2^N*

    }

    int count = 0;// *To count the number of valid subsets*

// *Iterate through all possible subsets*

*for* (int mask = 0; mask < totalSubsets; mask++)

    {

        int subsetSum = 0;

// *Calculate subset sum*

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

        {

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

            {// *If j-th bit is set, include S[j]*

                subsetSum += S[j];

            }

        }

// *If subset sum matches target, print the subset*

*if* (subsetSum == targetSum)

        {

            cout << "{ ";

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

            {

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

                {

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

                }

            }

            cout << "}" << endl;

            count++;// *Increase count of valid subsets*

        }

    }

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

*return* 0;

}

**Input**

Enter the number of elements: 6

Enter the elements: 5 10 12 13 15 18

Enter the target sum: 30

**Output**

{ 5 12 13 }

{ 5 10 15 }

{ 12 18 }

Total subsets found: 3

**Problem Number: 9**

**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

**Logic:** Imagine a thief breaking into a house with a knapsack of limited capacity. The house has valuable items, each with a specific weight and value. The thief must decide which items to take without exceeding the weightlimit. However, he cannot split items—he must either take an item fully or leave it behind.

**Algorithm**

**Step 1 -** Input the values: Read the number of items n, the maximum knapsack capacity W, and the arrays for item values val[] and weights wt[].

**Step 2 -** Initialize the DP table: Create a dp[n+1][W+1] table initialized to 0, where dp[i][w] stores the maximum value possible using the first i items with weight limit w.

**Step 3 -** Fill the DP table: Iterate through each item i (from 1 to n) and each weight w (from 1 to W). If the item's weight wt[i-1] is less than or equal to w, update dp[i][w] as the maximum of either excluding the item (dp[i-1][w]) or including it (val[i-1] + dp[i-1][w - wt[i-1]]). If the item's weight exceeds w, set dp[i][w] = dp[i-1][w].

**Step 4 -** Extract the result: The final answer, representing the maximum value that can be obtained within the given weight limit, is stored in dp[n][W].

**Step 5 -** Return or print the result: Display the maximum value that can be carried in the knapsack.

**Source Code**

#*include* <bits/stdc++.h>

using namespace std;

int dp[100][10000];// *DP table*

int weight[100], profits[100];// *Arrays for weight and profits*

int main()

{

// *Taking the number of items*

    cout << "Enter number of items: ";

    int n;

    cin >> n;

// *Taking the profits of the items*

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

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

    {

        cin >> profits[i];

    }

// *Taking the weights of the items*

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

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

    {

        cin >> weight[i];

    }

// *Taking the knapsack capacity*

    int cap;

    cout << "Enter capacity: ";

    cin >> cap;

// *Initialize the dp table with 0*

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

    {

*for* (int w = 0; w <= cap; w++)

        {

            dp[i][w] = 0;

        }

    }

// *Fill the dp table (Tabulation approach)*

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

    {

*for* (int w = 1; w <= cap; w++)

        {

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

            {

                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weight[i - 1]] + profits[i - 1]);

            }

*else*

            {

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

            }

        }

    }

// *The maximum profit is stored in dp[n][cap]*

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

// *Now, we need to find the items that were included*

    cout << "Items included to achieve maximum profit:" << endl;

    int w = cap;

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

    {

// *If the item was included*

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

        {

            cout << "Item " << i << " (Profit: " << profits[i - 1] << ", Weight: " << weight[i - 1] << ")" << endl;

            w = w - weight[i - 1];// *Reduce the remaining capacity*

        }

    }

*return* 0;

}

**Input:**

Enter the profits of the items:

15 25 13 23

Enter the weights of the items:

2 6 12 9

Enter capacity: 20

**Output:**

Maximum profit in the knapsack: 63

Items included to achieve maximum profit:

Item 4 (Profit: 23, Weight: 9)

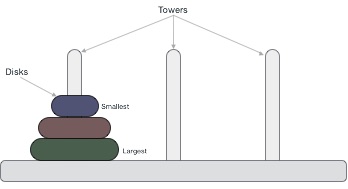
Item 2 (Profit: 25, Weight: 6)

Item 1 (Profit: 15, Weight: 2)

**Problem Number: 10**

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

Logic: Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –



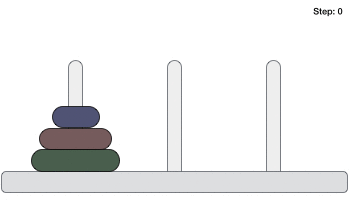
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

**Rules**

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

* Only one disk can be moved among the towers at any given time.
* Only the "top" disk can be removed.
* No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



Tower of Hanoi puzzle with n disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23 - 1 = 7** steps.

**Algorithm:**

**Step 1 -** First, we move the smaller (top) disk to aux peg.

**Step 2 -** Then, we move the larger (bottom) disk to destination peg.

**Step 3 -** And finally, we move the smaller disk from aux to destination peg.

**Source Code :**

#*include* <iostream>

using namespace std;

// *Function to solve the Tower of Hanoi problem*

void towerOfHanoi(int N, char from, char to, char aux) {

// *Base case: If only one disc, move it directly from source to destination*

*if* (N == 1) {

        cout << "Move disc 1 from " << from << " to " << to << endl;

*return*;

    }

// *Move N-1 discs from source to auxiliary peg*

    towerOfHanoi(N - 1, from, aux, to);

// *Move the remaining disc from source to destination*

    cout << "Move disc " << N << " from " << from << " to " << to << endl;

// *Move the N-1 discs from auxiliary peg to destination*

    towerOfHanoi(N - 1, aux, to, from);

}

int main() {

    int N;

    char source, destination, auxiliary;

// *Taking input for the number of discs*

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

    cin >> N;

// *Taking input for the names of the pegs (source, destination, auxiliary)*

    cout << "Enter the name of the source peg (e.g., A): ";

    cin >> source;

    cout << "Enter the name of the destination peg (e.g., C): ";

    cin >> destination;

    cout << "Enter the name of the auxiliary peg (e.g., B): ";

    cin >> auxiliary;

// *Call the function to solve Tower of Hanoi*

    towerOfHanoi(N, source, destination, auxiliary);

// *Calculate and display total number of moves*

    int totalMoves = (1 << N) - 1;// *Equivalent to 2^N - 1*

    cout << "Total number of moves: " << totalMoves << endl;

*return* 0;

}

**Input:**

Enter the number of discs: 3

Enter the name of the source peg (e.g., A): A

Enter the name of the destination peg (e.g., C): C

Enter the name of the auxiliary peg (e.g., B): B

**Output:**

Move disc 1 from A to C

Move disc 2 from A to B

Move disc 1 from C to B

Move disc 3 from A to C

Move disc 1 from B to A

Move disc 2 from B to C

Move disc 1 from A to C

Total number of moves: 7