**WEEK 1**

QUESTION NO.1:-Given a sorted array of positive integers containing few duplicate elements ,design an algorithm and implement it using a program to find whether given key element is present in the array or not.If present ,then also find the number of copies of given key.(T(n)=O(nlogn)).

Binary Search Algorithm:

The basic steps to perform Binary Search are: • Begin with an interval covering the whole array.

* If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
* Otherwise, narrow it to the upper half.
* Repeatedly check until the value is found or the interval is empty.

Code:

#include <stdio.h>

// Function to find the first or last occurrence of a given number in

// a sorted integer array. If `searchFirst` is true, return the

// first occurrence of the number; otherwise, return its last occurrence.

int binarySearch(int nums[], int n, int target, int searchFirst) {

// search space is nums[low…high]

int low = 0, high = n - 1;

// initialize the result by -1 int result = -1;

// loop till the search space is exhausted while (low <= high)

{

// find the mid-value in the search space and compares it with the target int mid = (low + high)/2;

// if the target is found, update the result if (target == nums[mid])

{

result = mid;

// go on searching towards the left (lower indices) if (searchFirst) {

high = mid - 1;

}

// go on searching towards the right (higher indices) else {

low = mid + 1;

}

}

// if the target is less than the middle element, discard the right half else if (target < nums[mid]) {

high = mid - 1;

}

// if the target is more than the middle element, discard the left half else {

low = mid + 1;

}

}

// return the found index or -1 if the element is not found return result;

}

int main(void)

{

int nums[] = {2, 5, 5, 5, 6, 6, 8, 9, 9, 9}; int target = 5;

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

// pass value 1 for the first occurrence

int first = binarySearch(nums, n, target, 1);

// pass value 0 for the last occurrence int last = binarySearch(nums, n, target, 0);

int count = last - first + 1;

if (first != -1) {

printf("Element %d occurs %d times", target, count);

} else {

printf("Element not found in the array");

}

return 0; }

Output:

5 occurs 3 times.

The time complexity of the above solution is O(log(n)) and doesn’t require any extra space, where n is the size of the input.

Ques 2:Given a sorted array of positive integers ,design an algorithm and implement it using a program to find three indices I,j,k such that a[i]+a[j]=a[k].

Algorithm:

The idea is similar to [Find a triplet that sum to a given value.](https://www.geeksforgeeks.org/find-a-triplet-that-sum-to-a-given-value/)  • Sort the given array first.

* Start fixing the greatest element of three from the back and traverse the array to find the other two numbers which sum up to the third element.
* Take two pointers j(from front) and k(initially i-1) to find the smallest of the two number and from i-1 to find the largest of the two remaining numbers
* If the addition of both the numbers is still less than A[i], then we need to increase the value of the summation of two numbers, thereby increasing the j pointer, so as to increase the value of **A[j] + A[k]**.
* If the addition of both the numbers is more than A[i], then we need to decrease the value of the summation of two numbers, thereby decrease the k pointer so as to decrease the overall value of **A[j] + A[k]**.

CODE:

#include<stdio.h>

// Utility function for finding

// triplet in array

void findTriplet(int arr[], int n)

{

// sort the array

sort(arr, arr + n);

// for every element in arr

// check if a pair exist(in array) whose // sum is equal to arr element for (int i = n - 1; i >= 0; i--) { int j = 0;

int k = i - 1;

// Iterate forward and backward to find

// the other two elements while (j < k) {

// If the two elements sum is // equal to the third element

if (arr[i] == arr[j] + arr[k]) {

// pair found

Printf( "numbers are " , arr[i] " " , arr[j] " " ,arr[k]); return;

}

// If the element is greater than

// sum of both the elements, then try

// adding a smaller number to reach the

// equality

else if (arr[i] > arr[j] + arr[k]) j += 1;

// If the element is smaller, then

// try with a smaller number // to reach equality, so decrease K else k -= 1;

} }

// No such triplet is found in array Printf( "No such triplet exists");

}

// driver program

int main()

{

int arr[] = { 5, 32, 1, 7, 10, 50, 19, 21, 2 }; int n = sizeof(arr) / sizeof(arr[0]);

findTriplet(arr, n); return 0;

}

Output: numbers are 21 2 19

**Time complexity**: O(N^2)

Ques 3:Given an array of non negative integers,design an algorithm and a program to count the number of integers such that their difference is equal to given key,K.

Algorithm:

1. Initialize count as 0
2. Sort all numbers in increasing order.
3. Remove duplicates from array.
4. Do following for each element arr[i]
   1. Binary Search for arr[i] + k in subarray from i+1 to n-1.
   2. If arr[i] + k found, increment count.
5. Return count.

Code:

#include <stdio.h>

/\* Standard binary search function \*/

int binarySearch(int arr[], int low, int high, int x)

{

if (high >= low)

{

int mid = low + (high - low)/2; if (x == arr[mid]) return mid; if (x > arr[mid])

return binarySearch(arr, (mid + 1), high, x); else return binarySearch(arr, low, (mid -1), x);

} return -1;

}

/\* Returns count of pairs with difference k in arr[] of size n. \*/ int countPairsWithDiffK(int arr[], int n, int k)

{

int count = 0, i;

sort(arr, arr+n); // Sort array elements

/\* code to remove duplicates from arr[] \*/

// Pick a first element point for (i = 0; i < n-1; i++) if (binarySearch(arr, i+1, n-1, arr[i] + k) != -1) count++;

return count;

}

// Driver program

int main()

{

int arr[] = {1, 5, 3, 4, 2}; int n = sizeof(arr)/sizeof(arr[0]); int k = 3;

printf("Count of pairs with given diff is " ,countPairsWithDiffK(arr, n, k)); return 0; }

**Output:**

Count of pairs with given diff is 2

Time complexity: The first step (sorting) takes O(nLogn) time. The second step runs binary search n times, so the time complexity of second step is also O(nLogn). Therefore, overall time complexity is O(nLogn).

**WEEK 2**

Ques 4:Given an unsorted array of integers,design an algorithm and a program to sort the array usin g selection sort.Your program should be able to find the no of comparisons and shifts required to sort the array.

Algorithm:

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list **Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

Code:

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

int temp = \*xp; \*xp = \*yp; \*yp = temp;

}

void selectionSort(int arr[], int n)

{

int i, j, min\_idx;

// One by one move boundary of unsorted subarray for (i = 0; i < n-1; i++)

{

// Find the minimum element in unsorted array min\_idx = i; for (j = i+1; j < n; j++) if (arr[j] < arr[min\_idx]) min\_idx = j;

// Swap the found minimum element with the first element swap(&arr[min\_idx], &arr[i]);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{ int i;

for (i=0; i < size; i++) printf("%d ", arr[i]); printf("\n");

}

// Driver program to test above functions int main()

{

int arr[] = {64, 25, 12, 22, 11}; int n = sizeof(arr)/sizeof(arr[0]); selectionSort(arr, n); printf("Sorted array: \n"); printArray(arr, n); return 0; }

**Output:**

Sorted array:

11 12 22 25 64

**Time Complexity:** O(n2) as there are two nested loops. **Auxiliary Space:** O(1)

Ques 5:Given an unsorted array of integers ,design an algorithm and implement a program to sort this array using selection sort.Your program should also find number of comparisons and number of swaps required.

Algorithm:

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

Code:

lst = [63, 42, 21, 9] lst.sort() list2 = [63, 42, 21, 9] a = 0 n=len(list2) for i in range(n):

min = i for j in range(i + 1, n): if lst == list2 :

break else : a += 1 if list2[j] < list2[min]:

min = j

list2[min], list2[i] = list2[i], list2[min]

print(list2)

print("Number of operation required :- ", a)

Output:

Number of Swap required by Selection sort: - 5

Time Complexity:O(n^2)

Ques 6:

Given an unsorted array of positive integers ,design an algorithm an implement it using a program to find whether there are any duplicate elements in the array or not.(T(n)=O(nlogn))

• **Algorithm:**

1. Traverse the given array from start to end.
2. For every element in the array increment the arr[i]%n‘th element by n.
3. Now traverse the array again and print all those indexes i for which arr[i]/n is greater than 1. Which guarantees that the number n has been added to that index
4. This approach works because all elements are in the range from 0 to n-1 and arr[i] would be greater than n only if a value “i” has appeared more than once

Code:

#include <stdio.h>

int main()

{

int numRay[] = { 0, 4, 3, 2, 7, 8, 2, 3, 1 }; int arr\_size = sizeof(numRay) / sizeof(numRay[0]);

// count the frequency for (int i = 0; i < arr\_size; i++) { numRay[numRay[i] % arr\_size]

= numRay[numRay[i] % arr\_size] + arr\_size;

}

printf("The repeating elements are : \n"); for (int i = 0; i < arr\_size; i++) { if (numRay[i] >= arr\_size \* 2) { printf("%d \n", i );

} } return 0; }

**Output:**

The repeating elements are :

2 3

**Complexity Analysis:**

• **Time Complexity:** O(n).

Only two traversals are needed. So the time complexity is O(n).

**WEEK 3**

Ques 7:

Given an unsorted Array of integers ,design an algorithm and implement it using a program to sort an array of elements by dividing the array into two subarrays and combining these subarrays after sorting each one of them.Your program should also find no. of comparisons and inversions during sorting the array.

Algorithm: **Simple Solution** is to sort the array using built in functions (generally an implementation of quick sort).

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

1. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
2. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Code:

#include <stdio.h> void mergeTwoHalf(int A[], int n)

{

// Sort the given array using sort STL sort(A, A + n);

}

// Driver code

int main()

{

int A[] = { 2, 3, 8, -1, 7, 10 }; int n = sizeof(A) / sizeof(A[0]); mergeTwoHalf(A, n);

// Print sorted Array

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

printf(“ “, A[i]);

return 0;

}

**Output**

-1 2 3 7 8 10

Time Complexity:O(n(logn))

Ques 8:Given an unsorted array of integers,design an algorithm and implement it using a program to sort an array of elements partioning the array into two sub arrays based on pivot element such that one of the sub array holds the value smaller than the pivot element and other holds value greater than the pivot element .Pivot element should be selected randomly from the array.Your program should also contain no of swaps and conversions required for the array.

Algorithm:

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

1. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
2. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Code:

#include <stdio.h>

// function to swap elements void swap(int \*a, int \*b) { int t = \*a; \*a = \*b;

\*b = t;

}

// function to find the partition position int partition(int array[], int low, int high) {

// select the rightmost element as pivot int pivot = array[high];

// pointer for greater element int i = (low - 1);

// traverse each element of the array

// compare them with the pivot

for (int j = low; j < high; j++) { if (array[j] <= pivot) {

// if element smaller than pivot is found // swap it with the greater element pointed by i i++;

// swap element at i with element at j

swap(&array[i], &array[j]);

}

}

// swap the pivot element with the greater element at i swap(&array[i + 1], &array[high]);

// return the partition point return (i + 1);

}

void quickSort(int array[], int low, int high) { if (low < high) {

// find the pivot element such that

// elements smaller than pivot are on left of pivot

// elements greater than pivot are on right of pivot int pi = partition(array, low, high);

// recursive call on the left of pivot quickSort(array, low, pi - 1);

// recursive call on the right of pivot quickSort(array, pi + 1, high);

}

}

// function to print array elements void printArray(int array[], int size) { for (int i = 0; i < size; ++i) { printf("%d ", array[i]);

}

printf("\n");

}

// main function int main() {

int data[] = {8, 7, 2, 1, 0, 9, 6};

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

printf("Unsorted Array\n"); printArray(data, n);

// perform quicksort on data quickSort(data, 0, n - 1);

printf("Sorted array in ascending order: \n"); printArray(data, n);

}

**Quicksort Complexity**

**Time Complexity**

Best O(n\*log n)

Worst O(n2)

Average O(n\*log n)

**Space Complexity** O(log n) **Stability** no

Ques 9:

Given an unsorted array of integers design an algorithm and implement it using a program to find Kth smallest or largest element in the array.(T(n)=O(n)).

Algorithm: 1) Sort the elements in descending order in O(n\*log(n)) 2) Print the first k numbers of the sorted array O(k).

Code:

#include <bits/stdc++.h>

using namespace std;

void kLargest(int arr[], int n, int k)

{

// Sort the given array arr in reverse // order.

sort(arr, arr + n, greater<int>());

// Print the first kth largest elements for (int i = 0; i < k; i++) cout << arr[i] << " ";

}

// driver program

int main()

{

int arr[] = { 1, 23, 12, 9, 30, 2, 50 }; int n = sizeof(arr) / sizeof(arr[0]); int k = 3;

kLargest(arr, n, k);

}

**Output**

50 30 23

**Time complexity:** O(n\*log(n)

**WEEK 4**

Ques 10:Given an unsorted array of alphabets containing duplicate elements.design an algorithm and implement it using a program to find which alphabet has maximum number of occurrence and print it.(T(n)=O(n)).

Algorithm: Linearly search for x, count the occurrences of x and return the count

Code:

#include<stdio.h>

// Returns number of times x occurs in arr[0..n-1] int countOccurrences(int arr[], int n, int x)

{

int res = 0; for (int i=0; i<n; i++) if (x == arr[i]) res++;

return res;

}

// Driver code

int main()

{

int arr[] = {1, 2, 2, 2, 2, 3, 4, 7 ,8 ,8 }; int n = sizeof(arr)/sizeof(arr[0]); int x = 2;

countOccurrences(arr, n, x); return 0; }

**Output :**

4

**Time Complexity:** O(n)

Ques 11:

Given an unsorted array of elements,design an algorithm and implement it using a program to find whether two elements exist such that their sum is equal to the given key element.(T(n)=O(nlog n))

**Algorithm:**

1. hasArrayTwoCandidates (A[], ar\_size, sum)
2. Sort the array in non-decreasing order.
3. Initialize two index variables to find the candidate elements in the sorted array.
   1. Initialize first to the leftmost index: l = 0
   2. Initialize second the rightmost index: r = ar\_size-1
4. Loop while l < r.
   1. If (A[l] + A[r] == sum) then return 1
   2. Else if( A[l] + A[r] < sum ) then l++
   3. Else r–
5. No candidates in the whole array – return 0

Code:

// C program to check if given array

// has 2 elements whose sum is equal

// to the given value

#include <stdio.h>

#define bool int

void quickSort(int\*, int, int);

bool hasArrayTwoCandidates(

int A[], int arr\_size, int sum)

{

int l, r;

/\* Sort the elements \*/

quickSort(A, 0, arr\_size - 1);

/\* Now look for the two candidates in the sorted array\*/ l = 0; r = arr\_size - 1; while (l < r) { if (A[l] + A[r] == sum) return 1; else if (A[l] + A[r] < sum) l++; else // A[i] + A[j] > sum r--; } return 0;

}

/\* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING PURPOSE \*/

void exchange(int\* a, int\* b)

{ int temp; temp = \*a;

\*a = \*b; \*b = temp;

}

int partition(int A[], int si, int ei)

{

int x = A[ei]; int i = (si - 1);

int j;

for (j = si; j <= ei - 1; j++) { if (A[j] <= x) { i++;

exchange(&A[i], &A[j]);

} }

exchange(&A[i + 1], &A[ei]); return (i + 1);

}

/\* Implementation of Quick Sort A[] --> Array to be sorted si --> Starting index

ei --> Ending index

\*/

void quickSort(int A[], int si, int ei)

{

int pi; /\* Partitioning index \*/ if (si < ei) {

pi = partition(A, si, ei); quickSort(A, si, pi - 1); quickSort(A, pi + 1, ei);

}

}

/\* Driver program to test above function \*/ int main()

{

int A[] = { 1, 4, 45, 6, 10, -8 }; int n = 16;

int arr\_size = 6;

if (hasArrayTwoCandidates(A, arr\_size, n)) printf("Array has two elements with given sum"); else

printf("Array doesn't have two elements with given sum");

getchar();

return 0;

}

**Output**

Array has two elements with given sum **Complexity Analysis:**

• **Time Complexity:** Depends on what sorting algorithm we use.

o If Merge Sort or Heap Sort is used then (-)(nlogn) in the worst case. o If Quick Sort is used then O(n^2) in the worst case.

Ques 12:

You have been given two second integer arrays of size m and n.design an algorithm and implement it using a program to find list of elements which are common to both.T(n)=O(m+n)

Algorithm:

1. Use two index variables i and j, initial values i = 0, j = 0
2. If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i. 3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.
3. If both are same then print any of them and increment both i and j.
4. Print remaining elements of the larger array.

Code:

#include <stdio.h>

/\* Function prints union of arr1[] and arr2[] m is the number of elements in arr1[]

n is the number of elements in arr2[] \*/

void printUnion(int arr1[], int arr2[], int m, int n)

{

int i = 0, j = 0; while (i < m && j < n) { if (arr1[i] < arr2[j]) printf(" %d ", arr1[i++]); else if (arr2[j] < arr1[i]) printf(" %d ", arr2[j++]); else {

printf(" %d ", arr2[j++]); i++;

} }

/\* Print remaining elements of the larger array \*/ while (i < m) printf(" %d ", arr1[i++]); while (j < n)

printf(" %d ", arr2[j++]);

}

/\* Driver program to test above function \*/ int main()

{

int arr1[] = { 1, 2, 4, 5, 6 }; int arr2[] = { 2, 3, 5, 7 }; int m = sizeof(arr1) / sizeof(arr1[0]); int n = sizeof(arr2) / sizeof(arr2[0]); printUnion(arr1, arr2, m, n); getchar();

return 0; }

Output:

1 2 3 4 5 6 7

**Time Complexity :** O(m + n)

**WEEK 5**

Ques 13: Given an unsorted array of alphabets containing duplicate elements. Design an algorithm and

implement it using a program to find which alphabet has maximum number of occurrences and

print it. (Time Complexity = O(n)) (Hint: Use counting sort)

Input Format:

The first line contains number of test cases, T.

For each test case, there will be two input lines.

First line contains n (the size of array).

Second line contains space-separated integers describing array.

Output:

The output will have T number of lines.

For each test case, output will be the array element which has maximum occurrences and its total

number of occurrences.

If no duplicates are present (i.e. all the elements occur only once), output should be “No

Duplicates Present”.

Algorithm:

1. Start
2. Declare the array.
3. Initialize the array.
4. Call the function that will return the most occurring element.
5. Declare two for loops
6. The first for loop will hold each element.
7. The second for loop will check for duplicate elements.
8. If duplicate elements found, increment the count.
9. If the count of the current element is more than the maximum count, then the maximum element is updated.
10. The maximum element counted is returned.
11. End.

Code:

#include<stdio.h>

//Program to count most occuring element

int getMaxRepeatingElement(int array[], int n)

{

int i, j, maxElement, count;

int maxCount = 0;

/\* Frequency of each element is counted and checked.If it's greater than the utmost count element we found till now, then it is updated accordingly \*/

for(i = 0; i< n; i++) //For loop to hold each element

{

count = 1;

for(j = i+1; j < n; j++) //For loop to check for duplicate elements

{

if(array[j] == array[i])

{

count++; //Increment count

/\* If count of current element is more than

maxCount, then update maxElement \*/

if(count > maxCount)

{

maxElement = array[j];

}

}

}

}

return maxElement;

}

//Driver Program

int main()

{

int n; //Array Size Declaration

printf("Enter the number of elements ");

scanf("%d",&n);

int array[n]; //Array Declaration

printf("Enter the array elements");

for(int i=0;i<n;i++) //Initializing Array Elements

{

scanf("%d",&array[i]);

}

int maxElement = getMaxRepeatingElement(array, n); //Function call

printf("\n Maximum Repeating Element : %d",maxElement); //Prints the most occuring element

return 0;

}

Enter the number of elements 5 Enter the array elements 2 4 5 3 5 Maximum Repeating Element: 5

**Program 2: Find the Maximum Repeating Element in an Array**

This is the most efficient method to find the number of most repeating elements in the array. The main concept behind using this approach is that if we sort the array, all the duplicate elements will get lined up next to each other. We can now linearly find the frequency of all elements in the array. This approach also ensures that the frequency is calculated only once for each unique element.

**Algorithm**

1. Start
2. Declare the array.
3. Initialize the array.
4. Call the function that will return the most occurring element.
5. Sort the array first.
6. Traverse the array to count the frequency of each element.
7. Return the element with the highest frequency.
8. Print the element.
9. End.

Below is the implementation in the C language.

#include<stdio.h>

#include<stdlib.h>

//Program to count most occuring element

int findMostFrequentElement(int A[], int n)

{

for (int i = 0; i < n; i++) //Sort the array

{

int temp;

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

{

if(A[i] > A[j])

{

temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

}

//finnd the most occuring element

int max\_count = 1, res = A[0], count = 1;

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

if (A[i] ==A[i - 1])

count++;

else {

if (count > max\_count) {

max\_count = count;

res = A[i - 1];

}

count = 1;

}

}

// If last element is most frequent

if (count > max\_count)

{

max\_count = count;

res = A[n - 1];

}

return res; //return the most repeatinng element

}

//Driver Program

int main()

{

int n; //Array Size Declaration

printf("Enter the number of elements ");

scanf("%d",&n);

int array[n]; //Array Declaration

printf("Enter the array elements");

for(int i=0;i<n;i++) //Initializing Array Elements

{

scanf("%d",&array[i]);

}

int maxElement = findMostFrequentElement(array, n); //Function call

printf("\n Maximum Repeating Element : %d",maxElement); //Prints the most occuring element

return 0;}

Output:

Enter the number of elements 5

Enter the array elements 2 4 5 3 5

Maximum Repeating Element: 5

Time Complexity:(O(n))

Ques 14: Given an unsorted array of integers, design an algorithm and implement it using a program to

find whether two elements exist such that their sum is equal to the given key element. (Time

Complexity = O(n log n)).

Input Format:

The first line contains number of test cases, T.

For each test case, there will be two input lines.

First line contains n (the size of array).

Second line contains space-separated integers describing array.

Third line contains key

Output Format:

The output will have T number of lines.

For each test case, output will be the elements arr[i] and arr[j] such that arr[i]+arr[j] = key if exist

otherwise print 'No Such Elements Exist”.

Algorithm:

Using brute force, A naive solution is to consider every pair in the given array and return if the desired sum is found. This approach is demonstrated below in C.

Code:

#include <stdio.h>

// Naive method to find a pair in an array with a given sum

void findPair(int nums[], int n, int target)

{

    // consider each element except the last

    for (int i = 0; i < n - 1; i++)

    {

        // start from the i'th element until the last element

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

        {

            // if the desired sum is found, print it

            if (nums[i] + nums[j] == target)

            {

                printf("Pair found (%d, %d)\n", nums[i], nums[j]);

                return;

            }

        }

    }

    // we reach here if the pair is not found

    printf("Pair not found");

}

int main(void)

{

    int nums[] = { 8, 7, 2, 5, 3, 1 };

    int target = 10;

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

    findPair(nums, n, target);

    return 0;

}

Output:

Pair found (8, 2)

Time Complexity:

The time complexity of the above solution is O(n2) and doesn’t require any extra space, where n is the size of the input.

Ques 15: You have been given two sorted integer arrays of size m and n. Design an algorithm and

implement it using a program to find list of elements which are common to both. (Time

Complexity = O(m+n))

Input Format:

First line contains m (the size of first array).

Second line contains m space-separated integers describing first array.

Third line contains n (the size of second array).

Fourth line contains n space-separated integers describing second array.

Output Format:

Output will be the list of elements which are common to both.

Algorithm:

For each element in A[], perform a linear search on B[]. Keep on adding the common elements to a list. To make sure the numbers in the list are unique, either check before adding new numbers to list or remove duplicates from the list after adding all numbers.

Pseudo Code:

int[] findIntersection(int A[], int B[], int m, int n)

{

int ans[]

for ( i = 0 to m-1 )

{

for ( j = 0 to n-1 )

{

if ( A[i] == B[j] )

if ( A[i] not in ans)

ans.add(A[i])

}

}

return ans

}

Output: 10 30 are the intersections.

Time Complexity: There are two loops in the solution where the outer loop runs n times and the inner loop runs m times. So in the worst case, time complexity = **O(m\*n)**.

**WEEK 6**

Ques 16: Given a (directed/undirected) graph, design an algorithm and implement it using a program to

find if a path exists between two given vertices or not. (Hint: use DFS)

Input Format:

Input will be the graph in the form of adjacency matrix or adjacency list.

Source vertex number and destination vertex number is also provided as an input.

Output Format:

Output will be 'Yes Path Exists' if path exists, otherwise print 'No Such Path Exists'.

Algorithm:

***BFS Algorithm:***

1. The implementation below is using BFS.
2. Create a queue and a visited array initially filled with 0, of size V where V is number of vertices.
3. Insert the starting node in the queue, i.e. push u in the queue and mark u as visited.
4. Run a loop until the queue is not empty.
5. Dequeue the front element of the queue. Iterate all its adjacent elements. If any of the adjacent element is the destination return true. Push all the adjacent and unvisited vertices in the queue and mark them as visited.
6. Return false as the destination is not reached in BFS.

Code:

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using adjacency list

// representation

class Graph

{

    int V;    // No. of vertices

    list<int> \*adj;    // Pointer to an array containing adjacency lists

public:

    Graph(int V);  // Constructor

    void addEdge(int v, int w); // function to add an edge to graph

    bool isReachable(int s, int d);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

// A BFS based function to check whether d is reachable from s.

bool Graph::isReachable(int s, int d)

{

    // Base case

    if (s == d)

      return true;

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    // Create a queue for BFS

    list<int> queue;

    // Mark the current node as visited and enqueue it

    visited[s] = true;

    queue.push\_back(s);

    // it will be used to get all adjacent vertices of a vertex

    list<int>::iterator i;

    while (!queue.empty())

    {

        // Dequeue a vertex from queue and print it

        s = queue.front();

        queue.pop\_front();

        // Get all adjacent vertices of the dequeued vertex s

        // If a adjacent has not been visited, then mark it visited

        // and enqueue it

        for (i = adj[s].begin(); i != adj[s].end(); ++i)

        {

            // If this adjacent node is the destination node, then

            // return true

            if (\*i == d)

                return true;

            // Else, continue to do BFS

            if (!visited[\*i])

            {

                visited[\*i] = true;

                queue.push\_back(\*i);

            }

        }

    }

    // If BFS is complete without visiting d

    return false;

}

// Driver program to test methods of graph class

int main()

{

    // Create a graph given in the above diagram

    Graph g(4);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(2, 3);

    g.addEdge(3, 3);

    int u = 1, v = 3;

    if(g.isReachable(u, v))

        cout<< "\n There is a path from " << u << " to " << v;

    else

        cout<< "\n There is no path from " << u << " to " << v;

    u = 3, v = 1;

    if(g.isReachable(u, v))

        cout<< "\n There is a path from " << u << " to " << v;

    else

        cout<< "\n There is no path from " << u << " to " << v;

    return 0;}

Output:

There is a path from 1 to 3

There is no path from 3 to 1

Time complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

Ques 17: Given a graph, design an algorithm and implement it using a program to find if a graph is

bipartite or not. (Hint: use BFS)

Input Format:

Input will be the graph in the form of adjacency matrix or adjacency list.

Output Format:

Output will be 'Yes Bipartite' if graph is bipartite, otherwise print 'Not Bipartite'.

Algorithm:

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).   
1. Assign RED color to the source vertex (putting into set U).   
2. Color all the neighbors with BLUE color (putting into set V).   
3. Color all neighbor’s neighbor with RED color (putting into set U).   
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.   
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Code:

#include <iostream>

#include <queue>

#define V 4

using namespace std;

// This function returns true if graph

// G[V][V] is Bipartite, else false

bool isBipartite(int G[][V], int src)

{

    // Create a color array to store colors

    // assigned to all vertices. Vertex

    // number is used as index in this array.

    // The value '-1' of colorArr[i]

    // is used to indicate that no color

    // is assigned to vertex 'i'. The value 1

    // is used to indicate first color

    // is assigned and value 0 indicates

    // second color is assigned.

    int colorArr[V];

    for (int i = 0; i < V; ++i)

        colorArr[i] = -1;

    // Assign first color to source

    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex

    // numbers and enqueue source vertex

    // for BFS traversal

    queue <int> q;

    q.push(src);

    // Run while there are vertices

    // in queue (Similar to BFS)

    while (!q.empty())

    {

        // Dequeue a vertex from queue ( Refer <http://goo.gl/35oz8> )

        int u = q.front();

        q.pop();

        // Return false if there is a self-loop

        if (G[u][u] == 1)

        return false;

        // Find all non-colored adjacent vertices

        for (int v = 0; v < V; ++v)

        {

            // An edge from u to v exists and

            // destination v is not colored

            if (G[u][v] && colorArr[v] == -1)

            {

                // Assign alternate color to this adjacent v of u

                colorArr[v] = 1 - colorArr[u];

                q.push(v);

            }

            // An edge from u to v exists and destination

            // v is colored with same color as u

            else if (G[u][v] && colorArr[v] == colorArr[u])

                return false;

        }

    }

    // If we reach here, then all adjacent

    // vertices can be colored with alternate color

    return true;

}

// Driver program to test above function

int main()

{

    int G[][V] = {{0, 1, 0, 1},

        {1, 0, 1, 0},

        {0, 1, 0, 1},

        {1, 0, 1, 0}

    };

    isBipartite(G, 0) ? cout << "Yes" : cout << "No";

    return 0;

}

Output:Yes.

Time Complexity:

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is O(V^2) where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes O(V+E).

**If Graph is represented using Adjacency List** .Time Complexity will be O(V+E).

Ques 18: Given a directed graph, design an algorithm and implement it using a program to find whether

cycle exists in the graph or not.

Input Format:

Input will be the graph in the form of adjacency matrix or adjacency list.

Output Format:

Output will be 'Yes Cycle Exists' if cycle exists otherwise print 'No Cycle Exists'.

* **Algorithm:**
  1. Create the graph using the given number of edges and vertices.
  2. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
  3. Mark the current node as visited and also mark the index in recursion stack.
  4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
  5. If the adjacent vertices are already marked in the recursion stack then return true.
  6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

Code:

|  |
| --- |
| #include<bits/stdc++.h>    using namespace std;    class Graph  {      int V;    // No. of vertices      list<int> \*adj;    // Pointer to an array containing adjacency lists      bool isCyclicUtil(int v, bool visited[], bool \*rs);  // used by isCyclic()  public:      Graph(int V);   // Constructor      void addEdge(int v, int w);   // to add an edge to graph      bool isCyclic();    // returns true if there is a cycle in this graph  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    // This function is a variation of DFSUtil() in <https://www.geeksforgeeks.org/archives/18212>  bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)  {      if(visited[v] == false)      {          // Mark the current node as visited and part of recursion stack          visited[v] = true;          recStack[v] = true;            // Recur for all the vertices adjacent to this vertex          list<int>::iterator i;          for(i = adj[v].begin(); i != adj[v].end(); ++i)          {              if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )                  return true;              else if (recStack[\*i])                  return true;          }        }      recStack[v] = false;  // remove the vertex from recursion stack      return false;  }    // Returns true if the graph contains a cycle, else false.  // This function is a variation of DFS() in <https://www.geeksforgeeks.org/archives/18212>  bool Graph::isCyclic()  {      // Mark all the vertices as not visited and not part of recursion      // stack      bool \*visited = new bool[V];      bool \*recStack = new bool[V];      for(int i = 0; i < V; i++)      {          visited[i] = false;          recStack[i] = false;      }        // Call the recursive helper function to detect cycle in different      // DFS trees      for(int i = 0; i < V; i++)          if ( !visited[i] && isCyclicUtil(i, visited, recStack))              return true;        return false;  }    int main()  {      // Create a graph given in the above diagram      Graph g(4);      g.addEdge(0, 1);      g.addEdge(0, 2);      g.addEdge(1, 2);      g.addEdge(2, 0);      g.addEdge(2, 3);      g.addEdge(3, 3);        if(g.isCyclic())          cout << "Graph contains cycle";      else          cout << "Graph doesn't contain cycle";      return 0;  } |

**Output:**

Graph contains cycle

* **Complexity Analysis:**
  + **Time Complexity:** O(V+E).   
    Time Complexity of this method is same as time complexity of [DFS traversal](https://www.geeksforgeeks.org/archives/18212) which is O(V+E).
  + **Space Complexity:** O(V).   
    To store the visited and recursion stack O(V) space is needed.

**WEEK 7**

Ques 19:

After end term examination, Akshay wants to party with his friends. All his friends are living as

paying guest and it has been decided to first gather at Akshay’s house and then move towards

party location. The problem is that no one knows the exact address of his house in the city.

Akshay as a computer science wizard knows how to apply his theory subjects in his real life and

came up with an amazing idea to help his friends. He draws a graph by looking in to location of

his house and his friends’ location (as a node in the graph) on a map. He wishes to find out

shortest distance and path covering that distance from each of his friend’s location to his house

and then whatsapp them this path so that they can reach his house in minimum time. Akshay has

developed the program that implements Dijkstra’s algorithm but not sure about correctness of

results. Can you also implement the same algorithm and verify the correctness of Akshay’s

results? (Hint: Print shortest path and distance from friends’ location to Akshay’s house)

Code:

#include <iostream>

#include<vector>

#include<bits/stdc++.h>

using namespace std;

int main() {

// your code goes here

int t;

cin>>t;

while(t--){

long long int n,a,b,c,cloth,i;

cin>>n>>a>>b>>c;

vector<int>f(n+1);

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

cin>>f[i];

}

f[n]=a;

sort(f.begin(),f.end());

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

if(f[i]==a){

if(b>a && b!=f[i+1]){

cloth=abs(b-f[i+1])+abs(f[i+1]-a);

}

else if(b<a && b!=f[i-1]){

cloth=abs(f[i-1]-b)+abs(a-f[i-1]);

}

else if(a==b){

if(i==0)

cloth=2\*(f[i+1]-a);

else

cloth=2\*min(abs(a-f[i-1]),abs(f[i+1]-a));

}

else{

cloth=abs(b-a);

}

break;

}

}

cout<<cloth+c<<endl;

}

return 0;

}

Time complexity: The time complexity is O(N)O(N)O(N) per test case.

Ques 20: Design an algorithm and implement it using a program to solve previous question's problem

using Bellman- Ford's shortest path algorithm.

Input Format:

Input will be the graph in the form of adjacency matrix or adjacency list.

Source vertex number is also provided as an input.

Output Format:

Output will contain V lines.

Each line will represent the whole path from destination vertex number to source vertex number

along with minimum path weigth.

**Algorithm**   
Following are the detailed steps.  
*Input:* Graph and a source vertex *src*   
*Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.  
**1)** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.  
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.   
…..**a)** Do following for each edge u-v   
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]   
………………….dist[v] = dist[u] + weight of edge uv  
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v   
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”   
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

Code:

|  |
| --- |
| #include <bits/stdc++.h>    // a structure to represent a weighted edge in graph  struct Edge {      int src, dest, weight;  };    // a structure to represent a connected, directed and  // weighted graph  struct Graph {      // V-> Number of vertices, E-> Number of edges      int V, E;        // graph is represented as an array of edges.      struct Edge\* edge;  };    // Creates a graph with V vertices and E edges  struct Graph\* createGraph(int V, int E)  {      struct Graph\* graph = new Graph;      graph->V = V;      graph->E = E;      graph->edge = new Edge[E];      return graph;  }    // A utility function used to print the solution  void printArr(int dist[], int n)  {      printf("Vertex   Distance from Source\n");      for (int i = 0; i < n; ++i)          printf("%d \t\t %d\n", i, dist[i]);  }    // The main function that finds shortest distances from src  // to all other vertices using Bellman-Ford algorithm.  The  // function also detects negative weight cycle  void BellmanFord(struct Graph\* graph, int src)  {      int V = graph->V;      int E = graph->E;      int dist[V];        // Step 1: Initialize distances from src to all other      // vertices as INFINITE      for (int i = 0; i < V; i++)          dist[i] = INT\_MAX;      dist[src] = 0;        // Step 2: Relax all edges |V| - 1 times. A simple      // shortest path from src to any other vertex can have      // at-most |V| - 1 edges      for (int i = 1; i <= V - 1; i++) {          for (int j = 0; j < E; j++) {              int u = graph->edge[j].src;              int v = graph->edge[j].dest;              int weight = graph->edge[j].weight;              if (dist[u] != INT\_MAX                  && dist[u] + weight < dist[v])                  dist[v] = dist[u] + weight;          }      }        // Step 3: check for negative-weight cycles.  The above      // step guarantees shortest distances if graph doesn't      // contain negative weight cycle.  If we get a shorter      // path, then there is a cycle.      for (int i = 0; i < E; i++) {          int u = graph->edge[i].src;          int v = graph->edge[i].dest;          int weight = graph->edge[i].weight;          if (dist[u] != INT\_MAX              && dist[u] + weight < dist[v]) {              printf("Graph contains negative weight cycle");              return; // If negative cycle is detected, simply                      // return          }      }        printArr(dist, V);        return;  }    // Driver program to test above functions  int main()  {      /\* Let us create the graph given in above example \*/      int V = 5; // Number of vertices in graph      int E = 8; // Number of edges in graph      struct Graph\* graph = createGraph(V, E);        // add edge 0-1 (or A-B in above figure)      graph->edge[0].src = 0;      graph->edge[0].dest = 1;      graph->edge[0].weight = -1;        // add edge 0-2 (or A-C in above figure)      graph->edge[1].src = 0;      graph->edge[1].dest = 2;      graph->edge[1].weight = 4;        // add edge 1-2 (or B-C in above figure)      graph->edge[2].src = 1;      graph->edge[2].dest = 2;      graph->edge[2].weight = 3;        // add edge 1-3 (or B-D in above figure)      graph->edge[3].src = 1;      graph->edge[3].dest = 3;      graph->edge[3].weight = 2;        // add edge 1-4 (or B-E in above figure)      graph->edge[4].src = 1;      graph->edge[4].dest = 4;      graph->edge[4].weight = 2;        // add edge 3-2 (or D-C in above figure)      graph->edge[5].src = 3;      graph->edge[5].dest = 2;      graph->edge[5].weight = 5;        // add edge 3-1 (or D-B in above figure)      graph->edge[6].src = 3;      graph->edge[6].dest = 1;      graph->edge[6].weight = 1;        // add edge 4-3 (or E-D in above figure)      graph->edge[7].src = 4;      graph->edge[7].dest = 3;      graph->edge[7].weight = -3;        BellmanFord(graph, 0);        return 0;  } |

**Output:**

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

Ques 21: Given a directed graph with two vertices ( source and destination). Design an algorithm and

implement it using a program to find the weight of the shortest path from source to destination

with exactly k edges on the path.

Input Format:

First input line will obtain number of vertices V present in the graph.

Graph in the form of adjacency matrix or adjacency list is taken as an input in next V lines.

Algorithm: The idea is to browse through all paths of length k from u to v using the approach discussed in the [previous post](https://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/) and return weight of the shortest path. A **simple solution** is to start from u, go to all adjacent vertices, and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following are C++ and Java implementations of this simple solution.

Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;    // Define number of vertices in the graph and infinite value  #define V 4  #define INF INT\_MAX    // A naive recursive function to count walks from u to v with k edges  int shortestPath(int graph[][V], int u, int v, int k)  {     // Base cases     if (k == 0 && u == v)             return 0;     if (k == 1 && graph[u][v] != INF) return graph[u][v];     if (k <= 0)                       return INF;       // Initialize result     int res = INF;       // Go to all adjacents of u and recur     for (int i = 0; i < V; i++)     {         if (graph[u][i] != INF && u != i && v != i)         {             int rec\_res = shortestPath(graph, i, v, k-1);             if (rec\_res != INF)                res = min(res, graph[u][i] + rec\_res);         }     }     return res;  }    // driver program to test above function  int main()  {      /\* Let us create the graph shown in above diagram\*/       int graph[V][V] = { {0, 10, 3, 2},                          {INF, 0, INF, 7},                          {INF, INF, 0, 6},                          {INF, INF, INF, 0}                        };      int u = 0, v = 3, k = 2;      cout << "Weight of the shortest path is " <<            shortestPath(graph, u, v, k);      return 0;  } |

**Output:**

Weight of the shortest path is 9

Time Complexity: The worst-case time complexity of the above function is O(Vk) where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly V children.

**WEEK 8**

Ques 22: Assume that a project of road construction to connect some cities is given to your friend. Map of

these cities and roads which will connect them (after construction) is provided to him in the form

of a graph. Certain amount of rupees is associated with construction of each road. Your friend

has to calculate the minimum budget required for this project. The budget should be designed in

such a way that the cost of connecting the cities should be minimum and number of roads

required to connect all the cities should be minimum (if there are N cities then only N-1 roads

need to be constructed). He asks you for help. Now, you have to help your friend by designing an

algorithm which will find minimum cost required to connect these cities. (use Prim's algorithm).

***Algorithm***   
**1)** Create a set *mstSet* that keeps track of vertices already included in MST.   
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.   
**3)** While mstSet doesn’t include all vertices   
….**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.   
….**b)** Include *u* to mstSet.   
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*  
The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Code:

|  |
| --- |
| #include <bits/stdc++.h>    using namespace std;    // Function to find out minimum valued node  // among the nodes which are not yet included in MST  int minnode(int n, int keyval[], bool mstset[]) {    int mini = numeric\_limits<int>::max();    int mini\_index;      // Loop through all the values of the nodes    // which are not yet included in MST and find    // the minimum valued one.    for (int i = 0; i < n; i++) {      if (mstset[i] == false && keyval[i] < mini) {        mini = keyval[i], mini\_index = i;      }    }    return mini\_index;  }    // Function to find out the MST and  // the cost of the MST.  void findcost(int n, vector<vector<int>> city) {      // Array to store the parent node of a    // particular node.    int parent[n];      // Array to store key value of each node.    int keyval[n];      // Boolean Array to hold bool values whether    // a node is included in MST or not.    bool mstset[n];      // Set all the key values to infinite and    // none of the nodes is included in MST.    for (int i = 0; i < n; i++) {      keyval[i] = numeric\_limits<int>::max();      mstset[i] = false;    }      // Start to find the MST from node 0.    // Parent of node 0 is none so set -1.    // key value or minimum cost to reach    // 0th node from 0th node is 0.    parent[0] = -1;    keyval[0] = 0;      // Find the rest n-1 nodes of MST.    for (int i = 0; i < n - 1; i++) {        // First find out the minimum node      // among the nodes which are not yet      // included in MST.      int u = minnode(n, keyval, mstset);        // Now the uth node is included in MST.      mstset[u] = true;        // Update the values of neighbor      // nodes of u which are not yet      // included in MST.      for (int v = 0; v < n; v++) {          if (city[u][v] && mstset[v] == false &&            city[u][v] < keyval[v]) {          keyval[v] = city[u][v];          parent[v] = u;        }      }    }      // Find out the cost by adding    // the edge values of MST.    int cost = 0;    for (int i = 1; i < n; i++)      cost += city[parent[i]][i];    cout << cost << endl;  }    // Utility Program:  int main() {      // Input 1    int n1 = 5;    vector<vector<int>> city1 = {{0, 1, 2, 3, 4},                                 {1, 0, 5, 0, 7},                                 {2, 5, 0, 6, 0},                                 {3, 0, 6, 0, 0},                                 {4, 7, 0, 0, 0}};    findcost(n1, city1);      // Input 2    int n2 = 6;    vector<vector<int>> city2 = {{0, 1, 1, 100, 0, 0},                                 {1, 0, 1, 0, 0, 0},                                 {1, 1, 0, 0, 0, 0},                                 {100, 0, 0, 0, 2, 2},                                 {0, 0, 0, 2, 0, 2},                                 {0, 0, 0, 2, 2, 0}};    findcost(n2, city2);      return 0;  } |

**Output:**

10

106

**Complexity:** The outer loop(i.e. the loop to add new node to MST) runs n times and in each iteration of the loop it takes O(n) time to find the minnode and O(n) time to update the neighboring nodes of u-th node. Hence the overall complexity is O(n2)

Ques 23: Implement the previous problem using Kruskal's algorithm.

Input Format:

The first line of input takes number of vertices in the graph.

Input will be the graph in the form of adjacency matrix or adjacency list.

Output Format:

Output will be minimum spanning weight

Algorithm:

Below are the steps for finding MST using Kruskal’s algorithm

**1.** Sort all the edges in non-decreasing order of their weight.   
**2.** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.   
**3.** Repeat step#2 until there are (V-1) edges in the spanning tree.

Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  // DSU data structure  //  path compression + rank by union    class DSU {      int\* parent;      int\* rank;    public:      DSU(int n)      {          parent = new int[n];          rank = new int[n];            for (int i = 0; i < n; i++) {              parent[i] = -1;              rank[i] = 1;          }      }        // Find function      int find(int i)      {          if (parent[i] == -1)              return i;            return parent[i] = find(parent[i]);      }      // union function      void unite(int x, int y)      {          int s1 = find(x);          int s2 = find(y);            if (s1 != s2) {              if (rank[s1] < rank[s2]) {                  parent[s1] = s2;                  rank[s2] += rank[s1];              }              else {                  parent[s2] = s1;                  rank[s1] += rank[s2];              }          }      }  };    class Graph {      vector<vector<int> > edgelist;      int V;    public:      Graph(int V) { this->V = V; }        void addEdge(int x, int y, int w)      {          edgelist.push\_back({ w, x, y });      }        void kruskals\_mst()      {          // 1. Sort all edges          sort(edgelist.begin(), edgelist.end());            // Initialize the DSU          DSU s(V);          int ans = 0;          cout << "Following are the edges in the "                  "constructed MST"               << endl;          for (auto edge : edgelist) {              int w = edge[0];              int x = edge[1];              int y = edge[2];                // take that edge in MST if it does form a cycle              if (s.find(x) != s.find(y)) {                  s.unite(x, y);                  ans += w;                  cout << x << " -- " << y << " == " << w                       << endl;              }          }          cout << "Minimum Cost Spanning Tree: " << ans;      }  };  int main()  {      /\* Let us create following weighted graph                     10                0--------1                |  \     |               6|   5\   |15                |      \ |                2--------3                    4       \*/      Graph g(4);      g.addEdge(0, 1, 10);      g.addEdge(1, 3, 15);      g.addEdge(2, 3, 4);      g.addEdge(2, 0, 6);      g.addEdge(0, 3, 5);        // int n, m;      // cin >> n >> m;        // Graph g(n);      // for (int i = 0; i < m; i++)      // {      //     int x, y, w;      //     cin >> x >> y >> w;      //     g.addEdge(x, y, w);      // }        g.kruskals\_mst();      return 0;  } |

**Output**

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

**Time Complexity:** O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

Ques 24: Assume that same road construction project is given to another person. The amount he will earn

from this project is directly proportional to the budget of the project. This person is greedy, so he

decided to maximize the budget by constructing those roads who have highest construction cost.

Design an algorithm and implement it using a program to find the maximum budget required for

the project.

Input Format:

The first line of input takes number of vertices in the graph.

Input will be the graph in the form of adjacency matrix or adjacency list.

Output Format:

Out will be maximum spanning weight.

Algorithm:

e can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.  
Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.  
cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

Code:

|  |
| --- |
| #include <bits/stdc++.h>  #include <iostream>  #include <math.h>  using namespace std;    // A utility function to get the maximum of two integers  int max(int a, int b) { return (a > b) ? a : b; }    /\* Returns the best obtainable price for a rod of length n     and price[] as prices of different pieces \*/  int cutRod(int price[], int index, int n)  {      // base case      if (index == 0) {          return n \* price[0];      }      //At any index we have 2 options either        //cut the rod of this length or not cut        //it      int notCut = cutRod(price,index - 1,n);      int cut = INT\_MIN;      int rod\_length = index + 1;        if (rod\_length <= n)          cut = price[index]                 + cutRod(price,index,n - rod\_length);        return max(notCut, cut);  }    /\* Driver program to test above functions \*/  int main()  {      int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };      int size = sizeof(arr) / sizeof(arr[0]);      cout << "Maximum Obtainable Value is "           << cutRod(arr, size - 1, size);      getchar();      return 0;  } |

**Output**

Maximum Obtainable Value is 22

**WEEK 9**

Ques 26: Given a graph, Design an algorithm and implement it using a program to implement Floyd-

Warshall all pair shortest path algorithm.

Input Format:

The first line of input takes number of vertices in the graph.

Input will be the graph in the form of adjacency matrix or adjacency list. If a direct edge is not

present between any pair of vertex (u,v), then this entry is shown as AdjM[u,v] = INF.

Output Format:

Output will be shortest distance matrix in the form of V X V matrix, where each entry (u,v)

represents shortest distance between vertex u and vertex v.

**Floyd Warshall Algorithm**   
We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.   
**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.   
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]  
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.  
 Code:

|  |
| --- |
| #include<stdio.h>    // Number of vertices in the graph  #define V 4    /\* Define Infinite as a large enough    value. This value will be used    for vertices not connected to each other \*/  #define INF 99999    // A function to print the solution matrix  void printSolution(int dist[][V]);    // Solves the all-pairs shortest path  // problem using Floyd Warshall algorithm  void floydWarshall (int graph[][V])  {      /\* dist[][] will be the output matrix        that will finally have the shortest        distances between every pair of vertices \*/      int dist[V][V], i, j, k;        /\* Initialize the solution matrix        same as input graph matrix. Or         we can say the initial values of         shortest distances are based         on shortest paths considering no         intermediate vertex. \*/      for (i = 0; i < V; i++)          for (j = 0; j < V; j++)              dist[i][j] = graph[i][j];        /\* Add all vertices one by one to        the set of intermediate vertices.        ---> Before start of an iteration, we        have shortest distances between all        pairs of vertices such that the shortest        distances consider only the        vertices in set {0, 1, 2, .. k-1} as        intermediate vertices.        ----> After the end of an iteration,        vertex no. k is added to the set of        intermediate vertices and the set        becomes {0, 1, 2, .. k} \*/      for (k = 0; k < V; k++)      {          // Pick all vertices as source one by one          for (i = 0; i < V; i++)          {              // Pick all vertices as destination for the              // above picked source              for (j = 0; j < V; j++)              {                  // If vertex k is on the shortest path from                  // i to j, then update the value of dist[i][j]                  if (dist[i][k] + dist[k][j] < dist[i][j])                      dist[i][j] = dist[i][k] + dist[k][j];              }          }      }        // Print the shortest distance matrix      printSolution(dist);  }    /\* A utility function to print solution \*/  void printSolution(int dist[][V])  {      printf ("The following matrix shows the shortest distances"              " between every pair of vertices \n");      for (int i = 0; i < V; i++)      {          for (int j = 0; j < V; j++)          {              if (dist[i][j] == INF)                  printf("%7s", "INF");              else                  printf ("%7d", dist[i][j]);          }          printf("\n");      }  }    // driver program to test above function  int main()  {      /\* Let us create the following weighted graph              10         (0)------->(3)          |         /|\        5 |          |          |          | 1         \|/         |         (1)------->(2)              3           \*/      int graph[V][V] = { {0,   5,  INF, 10},                          {INF, 0,   3, INF},                          {INF, INF, 0,   1},                          {INF, INF, INF, 0}                        };        // Print the solution      floydWarshall(graph);      return 0;  } |

**Output:**

Following matrix shows the shortest distances between every pair of vertices

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Time Complexity:** O(V^3)  
The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.   
Also, the value of INF can be taken as INT\_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT\_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

Ques 27: Given a knapsack of maximum capacity w. N items are provided, each having its own value and

weight. You have to Design an algorithm and implement it using a program to find the list of the

selected items such that the final selected content has weight w and has maximum value. You can

take fractions of items,i.e. the items can be broken into smaller pieces so that you have to carry.

**Method 1:** Recursion by Brute-Force algorithm OR Exhaustive Search.  
**Approach:** A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.  
***Optimal Sub-structure*:** To consider all subsets of items, there can be two cases for every item.

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and **Case 1** is the only possibility.

Code:

|  |
| --- |
| #include <stdio.h>    // A utility function that returns  // maximum of two integers  int max(int a, int b) { return (a > b) ? a : b; }    // Returns the maximum value that can be  // put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {      // Base Case      if (n == 0 || W == 0)          return 0;        // If weight of the nth item is more than      // Knapsack capacity W, then this item cannot      // be included in the optimal solution      if (wt[n - 1] > W)          return knapSack(W, wt, val, n - 1);        // Return the maximum of two cases:      // (1) nth item included      // (2) not included      else          return max(              val[n - 1]                  + knapSack(W - wt[n - 1],                             wt, val, n - 1),              knapSack(W, wt, val, n - 1));  }    // Driver program to test above function  int main()  {      int val[] = { 60, 100, 120 };      int wt[] = { 10, 20, 30 };      int W = 50;      int n = sizeof(val) / sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

**Output**

220

It should be noted that the above function computes the same sub-problems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. The time complexity of this naive recursive solution is exponential (2^n).

Ques 28:Given an array of elements.assume arr[i] represents size of array.write an algo and program to merge these files with minimum computation .

Algorithm:

Node represents a file with a given size also given nodes are greater than 2

1. Add all the nodes in a priority queue (Min Heap).{node.weight = file size}
2. Initialize count = 0 // variable to store file computations.
3. Repeat while (size of priority Queue is greater than 1)
   1. create a new node
   2. new node = pq.poll().weight+pq.poll().weight;//pq denotes priority queue, remove 1st smallest and 2nd smallest element and add their weights to get a new node
   3. count += node.weight
   4. add this new node to priority queue;
4. count is the final answer

Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;    // Function to find minimum computation  int minComputation(int size, int files[])  {        // Create a min heap      priority\_queue<int, vector<int>, greater<int> > pq;        for (int i = 0; i < size; i++) {            // Add sizes to priorityQueue          pq.push(files[i]);      }        // Variable to count total Computation      int count = 0;        while (pq.size() > 1) {            // pop two smallest size element          // from the min heap          int first\_smallest = pq.top();          pq.pop();          int second\_smallest = pq.top();          pq.pop();            int temp = first\_smallest + second\_smallest;            // Add the current computations          // with the previous one's          count += temp;            // Add new combined file size          // to priority queue or min heap          pq.push(temp);      }      return count;  }    // Driver code  int main()  {        // No of files      int n = 6;        // 6 files with their sizes      int files[] = { 2, 3, 4, 5, 6, 7 };        // Total no of computations      // do be done final answer      cout << "Minimum Computations = "           << minComputation(n, files);        return 0;  }    // This code is contributed by jaigoyal1328 |

**Output**

Minimum Computations = 68

**WEEK 10**

Ques 29: Given a list of activities with their starting time and finishing time. Your goal is to select

maximum number of activities that can be performed by a single person such that selected

activities must be non-conflicting. Any activity is said to be non-conflicting if starting time of an

activity is greater than or equal to the finishing time of the other activity. Assume that a person

can only work on a single activity at a time.

Input Format:

First line of input will take number of activities N.

Second line will take N space-separated values defining starting time for all the N activities.

Third line of input will take N space-separated values defining finishing time for all the N

activities.

Output Format:

Output will be the number of non-conflicting activities and the list of selected activities.

Algorithm:

* 1. Sort the activities according to their finishing time   
     2) Select the first activity from the sorted array and print it.   
     3) Do the following for the remaining activities in the sorted array.   
     …….a) If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.  
     In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

Code:

|  |
| --- |
| #include<stdio.h>    // Prints a maximum set of activities that can be done by a single  // person, one at a time.  //  n   -->  Total number of activities  //  s[] -->  An array that contains start time of all activities  //  f[] -->  An array that contains finish time of all activities  void printMaxActivities(int s[], int f[], int n)  {      int i, j;        printf ("Following activities are selected n");        // The first activity always gets selected      i = 0;      printf("%d ", i);        // Consider rest of the activities      for (j = 1; j < n; j++)      {        // If this activity has start time greater than or        // equal to the finish time of previously selected        // activity, then select it        if (s[j] >= f[i])        {            printf ("%d ", j);            i = j;        }      }  }    // driver program to test above function  int main()  {      int s[] =  {1, 3, 0, 5, 8, 5};      int f[] =  {2, 4, 6, 7, 9, 9};      int n = sizeof(s)/sizeof(s[0]);      printMaxActivities(s, f, n);      return 0;  } |

**Output**

Following activities are selected n0 1 3 4

**How does Greedy Choice work for Activities sorted according to finish time?**   
Let the given set of activities be S = {1, 2, 3, …n} and activities are sorted by finish time. The greedy choice is to always pick activity 1. How come activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with the first activity other than 1, then there is also a solution A of the same size with activity 1 as the first activity. Let the first activity selected by B be k, then there always exist A = {B – {k}} U {1}.

Ques 30: Given a long list of tasks. Each task takes specific time to accomplish it and each task has a

deadline associated with it. You have to design an algorithm and implement it using a program to

find maximum number of tasks that can be completed without crossing their deadlines and also

find list of selected tasks.

Input Format:

First line will give total number of tasks n.

Second line of input will give n space-separated elements of array representing time taken by

each task.

Third line of input will give n space-separated elements of array representing deadline associated

with each task.

Output Format:

Output will be the total number of maximum tasks that can be completed.

Algorithm:

* 1. Sort all jobs in decreasing order of profit.   
     2) Iterate on jobs in decreasing order of profit.For each job , do the following :   
     a)Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in   
     this slot and mark this slot filled.   
     b)If no such i exists, then ignore the job.

Code:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    // A structure to represent a job  typedef struct Job {      char id; // Job Id      int dead; // Deadline of job      int profit; // Profit if job is over before or on deadline  } Job;    // This function is used for sorting all jobs according to  // profit  int compare(const void\* a, const void\* b)  {      Job\* temp1 = (Job\*)a;      Job\* temp2 = (Job\*)b;      return (temp2->profit - temp1->profit);  }    // Find minimum between two numbers.  int min(int num1, int num2)  {      return (num1 > num2) ? num2 : num1;  }    // Returns minimum number of platforms required  void printJobScheduling(Job arr[], int n)  {      // Sort all jobs according to decreasing order of profit      qsort(arr, n, sizeof(Job), compare);      //     sort(arr, arr+n, comparison);        int result[n]; // To store result (Sequence of jobs)      bool slot[n]; // To keep track of free time slots        // Initialize all slots to be free      for (int i = 0; i < n; i++)          slot[i] = false;        // Iterate through all given jobs      for (int i = 0; i < n; i++) {          // Find a free slot for this job (Note that we start          // from the last possible slot)          for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {              // Free slot found              if (slot[j] == false) {                  result[j] = i; // Add this job to result                  slot[j] = true; // Make this slot occupied                  break;              }          }      }        // Print the result      for (int i = 0; i < n; i++)          if (slot[i])              printf("%c ", arr[result[i]].id);  }    // Driver code  int main()  {      Job arr[] = { { 'a', 2, 100 },                    { 'b', 1, 19 },                    { 'c', 2, 27 },                    { 'd', 1, 25 },                    { 'e', 3, 15 } };      int n = sizeof(arr) / sizeof(arr[0]);      printf(          "Following is maximum profit sequence of jobs \n");        // Function call      printJobScheduling(arr, n);      return 0;  } |

**Output**

Following is maximum profit sequence of jobs

c a e

The **Time Complexity** of the above solution is O(n2). It can be optimized using **Priority Queue(max heap)**.

Ques 31: Given an unsorted array of elements, design an algorithm and implement it using a program to

find whether majority element exists or not. Also find median of the array. A majority element is

an element that appears more than n/2 times, where n is the size of array.

Input Format:

First line of input will give size n of array.

Second line of input will take n space-separated elements of array.

Output Format:

First line of output will be 'yes' if majority element exists, otherwise print 'no'.

Second line of output will print median of the array.

 **Algorithm:**

1. Create a variable to store the max count, *count = 0*
2. Traverse through the array from start to end.
3. For every element in the array run another loop to find the count of similar elements in the given array.
4. If the count is greater than the max count update the max count and store the index in another variable.
5. If the maximum count is greater than the half the size of the array, print the element. Else print there is no majority element.

Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;    // Function to find Majority element  // in an array  void findMajority(int arr[], int n)  {      int maxCount = 0;      int index = -1; // sentinels      for (int i = 0; i < n; i++) {          int count = 0;          for (int j = 0; j < n; j++) {              if (arr[i] == arr[j])                  count++;          }            // update maxCount if count of          // current element is greater          if (count > maxCount) {              maxCount = count;              index = i;          }      }        // if maxCount is greater than n/2      // return the corresponding element      if (maxCount > n / 2)          cout << arr[index] << endl;        else          cout << "No Majority Element" << endl;  }    // Driver code  int main()  {      int arr[] = { 1, 1, 2, 1, 3, 5, 1 };      int n = sizeof(arr) / sizeof(arr[0]);        // Function calling      findMajority(arr, n);        return 0;  } |

**Output**

1

**Complexity Analysis:**

* **Time Complexity:** O(n\*n).   
  A nested loop is needed where both the loops traverse the array from start to end, so the time complexity is O(n^2).
* **Auxiliary Space:** O(1).   
  As no extra space is required for any operation so the space complexity is constant.

**WEEK 11**

Ques 32: Given a sequence of matrices, write an algorithm to find most efficient way to multiply these

matrices together. To find the optimal solution, you need to find the order in which these

matrices should be multiplied.

Input Format:

First line of input will take number of matrices n that you need to multiply.

For each line i in n, take two inputs which will represent dimensions aXb of matrix i.

Output Format:

Output will be the minimum number of operations that are required to multiply the list of

matrices.

Algorithm:

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.  
Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int dp[100][100];    // Function for matrix chain multiplication  int matrixChainMemoised(int\* p, int i, int j)  {      if (i == j)      {          return 0;      }      if (dp[i][j] != -1)      {          return dp[i][j];      }      dp[i][j] = INT\_MAX;      for (int k = i; k < j; k++)      {          dp[i][j] = min(              dp[i][j], matrixChainMemoised(p, i, k)                       + matrixChainMemoised(p, k + 1, j)                         + p[i - 1] \* p[k] \* p[j]);      }      return dp[i][j];  }  int MatrixChainOrder(int\* p, int n)  {      int i = 1, j = n - 1;      return matrixChainMemoised(p, i, j);  }    // Driver Code  int main()  {      int arr[] = { 1, 2, 3, 4 };      int n = sizeof(arr) / sizeof(arr[0]);      memset(dp, -1, sizeof dp);        cout << "Minimum number of multiplications is "           << MatrixChainOrder(arr, n);  } |

**Output**

Minimum number of multiplications is 18

**Time Complexity:** O(n3 )

**Auxiliary Space:** O(n2) ignoring recursion stack space

Ques 33:

Given a set of available types of coins. Let suppose you have infinite supply of each type of coin.

For a given value N, you have to Design an algorithm and implement it using a program to find

number of ways in which these coins can be added to make sum value equals to N.

Input Format:

First line of input will take number of coins that are available.

Second line of input will take the value of each coin.

Third line of input will take the value N for which you need to find sum.

Output Format:

Output will be the number of ways.

**3) Approach (Algorithm)**

See, here each coin of a given denomination can come an infinite number of times. (Repetition allowed), this is what we call UNBOUNDED KNAPSACK. We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.  But here, the inclusion process is not for just once; we can include any denomination any number of times until N<0.

Basically, If we are at s[m-1], we can take as many instances of that coin ( unbounded inclusion ) i.e **count(S, m, n – S[m-1] )** ; then we move to s[m-2]. After moving to s[m-2], we can’t move back and can’t make choices for s[m-1] i.e **count(S, m-1, n )**.

Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(S, m, n – S[m-1] ) + count(S, m-1, n ) ;** which will be our required answer.

Code:

|  |
| --- |
| #include<stdio.h>    // Returns the count of ways we can  // sum S[0...m-1] coins to get sum n  int count( int S[], int m, int n )  {      // If n is 0 then there is 1 solution      // (do not include any coin)      if (n == 0)          return 1;        // If n is less than 0 then no      // solution exists      if (n < 0)          return 0;        // If there are no coins and n      // is greater than 0, then no      // solution exist      if (m <=0 && n >= 1)          return 0;        // count is sum of solutions (i)      // including S[m-1] (ii) excluding S[m-1]      return count( S, m - 1, n ) + count( S, m, n-S[m-1] );  }    // Driver program to test above function  int main()  {      int i, j;      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      printf("%d ", count(arr, m, 4));      getchar();      return 0;  } |

**Output**

4

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.

Ques 34: Given a set of elements, you have to partition the set into two subsets such that the sum of

elements in both subsets is same. Design an algorithm and implement it using a program to solve

this problem.

Input Format:

First line of input will take number of elements n present in the set.

Second line of input will take n space-separated elements of the set.

Output Format:

Output will be 'yes' if two such subsets found otherwise print 'no'.

Algorithm:

Following are the two main steps to solve this problem:   
1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.   
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.   
The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Code:

|  |
| --- |
| #include <stdbool.h>  #include <stdio.h>    // A utility function that returns true if there is  // a subset of arr[] with sum equal to given sum  bool isSubsetSum(int arr[], int n, int sum)  {      // Base Cases      if (sum == 0)          return true;      if (n == 0 && sum != 0)          return false;        // If last element is greater than sum, then      // ignore it      if (arr[n - 1] > sum)          return isSubsetSum(arr, n - 1, sum);        /\* else, check if sum can be obtained by any of         the following         (a) including the last element         (b) excluding the last element      \*/      return isSubsetSum(arr, n - 1, sum)             || isSubsetSum(arr, n - 1, sum - arr[n - 1]);  }    // Returns true if arr[] can be partitioned in two  //  subsets of equal sum, otherwise false  bool findPartiion(int arr[], int n)  {      // Calculate sum of the elements in array      int sum = 0;      for (int i = 0; i < n; i++)          sum += arr[i];        // If sum is odd, there cannot be two subsets      // with equal sum      if (sum % 2 != 0)          return false;        // Find if there is subset with sum equal to      // half of total sum      return isSubsetSum(arr, n, sum / 2);  }    // Driver code  int main()  {      int arr[] = { 3, 1, 5, 9, 12 };      int n = sizeof(arr) / sizeof(arr[0]);        // Function call      if (findPartiion(arr, n) == true)          printf("Can be divided into two subsets "                 "of equal sum");      else          printf("Can not be divided into two subsets"                 " of equal sum");      return 0;  } |

**Output**

Can be divided into two subsets of equal sum

**Time Complexity:** O(2^n) In the worst case, this solution tries two possibilities (whether to include or exclude) for every element.