#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print\_vector(int v[], int n) {

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

cout << v[i] << " ";

}

cout << endl;

}

//merge sort int

void merge(int arr[], int p, int q, int r) {

// Crea le grandezze dei sottoarray L ← arr[p..q] and M ← arr[q+1..r]

int n1 = q - p + 1;

int n2 = r - q;

//int L[n1], M[n2];

int L[n1], M[n2];

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

L[i] = arr[p + i];

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

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

// Indici dei sotto array e dell'array principale

int i, j, k;

i = 0;

j = 0;

k = p;

//ordiniamo gli elementi di L e M fino alle estremità

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

if (L[i] <= M[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = M[j];

j++;

}

k++;

}

//al terminare degli elementi di uno dei due sotto array prendo tutti gli elementi dell'altro

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = M[j];

j++;

k++;

}

}

// divide l'array in due sottoarray, li ordina e ne fa il merge

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

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

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

{

int result = 0;

if ((low > high) || (low == high && arr[low] != x))

return 0;

if (low == high && arr[low] == x)

return 1;

result = binarysearch(arr, low, (low + high) / 2, x) + binarysearch(arr, 1 + (low + high) / 2,high, x);

return result;

}

void algorithm(int v[], int dim, int elem)

{

//ordino l'array con il merge sort O(nlogn)

//mergeSort(v, 0, dim - 1);

print\_vector(v, dim);

//cerco con una ricerca binaria modificata il numero di occorrenze

cout << binarysearch(v, 0, dim - 1, elem);

}

// input numero di test case numero di elementi poi elementi

void input()

{

int a = 0;

cin >> a;

while (a)

{

int b = 0;

int dim = 0;

cin >> b;

cin >> dim;

int v[dim];

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

{

cin >> v[i];

}

algorithm(v, dim, b);

a--;

}

}

int main()

{

input();

return 0;

}

Analisi della complessità: La complessità è O(logn) della mia ricerca binaria modificata.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

//funzione che controlla se il numero è primo

bool isPrime(int n)

{

if (n == 2)

return true;

if (n % 2 == 0)

return false;

for (int i = 3; i \* i <= n; i += 2)

{

if (n % i == 0)

return false;

}

return true;

}

//funzione di backtracking

void find\_sum\_backtracking(int prime, int target, int curr, int count, vector<int> &curr\_solution, vector<vector<int>> &solution)

{

if (curr == target)

{

//mi salvo la soluzione trovata

solution.push\_back(curr\_solution);

return;

}

if (curr > target)

return;

for (int i = prime + 1; i <= target; i++)

{

//verifico se il numero che ottengo dal ciclo è primo

if (isPrime(i))

{

curr\_solution.push\_back(i);

find\_sum\_backtracking(i, target, curr + i, count + 1, curr\_solution, solution);

curr\_solution.pop\_back();

}

}

}

vector<vector<int>> find\_sum(int prime, int target\_sum, int count)

{

vector<vector<int>> solution;

vector<int> curr\_solution;

find\_sum\_backtracking(prime, target\_sum, 0, count, curr\_solution, solution);

return solution;

}

int main()

{

int n;

cin >> n;

int i = 1;

while (n--)

{

int a;

int b;

int c;

cin >> a;

cin >> b;

cin >> c;

vector<vector<int>> solution = find\_sum(c, b, a);

cout << "CASO DI TEST " << i << endl;

// stampo la soluzione dalla matrice delle soluzioni

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

{

for (int j = 0; j < solution[i].size(); j++)

{

cout << solution[i][j] << " ";

}

cout << endl;

}

++i;

}

return 0;

}

Analisi della complessità: l’approccio di backtracking tipicamente ha una complessità esponenziale o fattoriale poiché è un approccio brute force O(m\*n!).