**LIS:**

#include <bits/stdc++.h>

**using** **namespace** std;

#define ll long long int

**int** find\_lis(vector<**int**> a)

{

vector<**int**> dp;

**for** (**int** i : a)

{

**int** pos = lower\_bound(dp.begin(), dp.end(), i) - dp.begin();

**if** (pos == dp.size())

{

dp.push\_back(i);

}

**else**

{

dp[pos] = i;

}

}

**return** dp.size();

}

**int** main()

{

**int** n;

cin >> n;

vector<**int**> num(n);

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

{

cin >> num[i];

}

**int** ans = find\_lis(num);

cout << ans << endl;

**return** 0;

}

**0/1 Knapsack:**

#include <iostream>

#include <vector>

**using** **namespace** std;

#define ll long long int

**int** main()

{

**int** n, ks;

cin >> n >> ks;

vector<**int**> item(n), price(n);

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

{

cin >> item[i];

}

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

{

cin >> price[i];

}

vector<vector<**int**>> ans(n + 1, vector<**int**>(ks + 1, 0));

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

{

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

{

**if** (j < item[i - 1])

{

ans[i][j] = ans[i - 1][j];

}

**else**

{

ans[i][j] = max(ans[i - 1][j], ans[i - 1][j - item[i - 1]] + price[i - 1]);

}

}

}

ll cnt = 0;

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

{

cnt += price[i];

}

**if** (cnt == ans[n][ks])

{

cout << "My King, I am successful in capturing the big fish. Immortality is few steps away." << endl;

}

**else**

{

cout << "My King, I have captured " << ans[n][ks] << " followers till now and I need more soldiers asap." << endl;

}

**return** 0;

}

**BFS Traversal:**

#include <bits/stdc++.h>

**using** **namespace** std;

**void** bfs(vector<vector<**int**>> &adjMatrix, vector<**bool**> &visited, vector<**int**> &ans, **int** vertex, **int** start)

{

queue<**int**> q;

q.push(start);

visited[start] = **true**;

**while** (!q.empty())

{

**int** frontNode = q.front();

q.pop();

ans.push\_back(frontNode);

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

{

**if** (adjMatrix[frontNode][i] == 1 && !visited[i])

{

q.push(i);

visited[i] = **true**;

}

}

}

}

vector<**int**> BFS(vector<vector<**int**>> &adjMatrix, **int** vertex, **int** start)

{

vector<**int**> ans;

vector<**bool**> visited(vertex+1, **false**);

bfs(adjMatrix, visited, ans, vertex, start);

**return** ans;

}

**int** main()

{

**int** vertex;

cin >> vertex;

vector<vector<**int**>> adjMatrix(vertex+1, vector<**int**>(vertex+1, 0));

// Input the adjacency matrix

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

{

**for** (**int** j = 1; j <= vertex; j++)

{

cin >> adjMatrix[i][j];

}

}

**int** start;

cout << "Enter the starting point for BFS: ";

cin >> start;

vector<**int**> ans = BFS(adjMatrix, vertex, start);

cout << "BFS traversal from " << start << ": ";

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

{

cout << ans[i] << " ";

}

cout << endl;

**return** 0;

}

**Closet Pair of Points:**

// A divide and conquer program in C++

// to find the smallest distance from a

// given set of points.

#include <bits/stdc++.h>

**using** **namespace** std;

**class** Point

{

**public**:

**int** x, y;

};

**int** compareX(**const** **void** \*a, **const** **void** \*b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

**return** (p1->x - p2->x);

}

**int** compareY(**const** **void** \*a, **const** **void** \*b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

**return** (p1->y - p2->y);

}

**float** dist(Point p1, Point p2)

{

**return** sqrt((p1.x - p2.x) \* (p1.x - p2.x) +

(p1.y - p2.y) \* (p1.y - p2.y));

}

**float** bruteForce(Point P[], **int** n)

{

**float** min = FLT\_MAX;

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

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

**if** (dist(P[i], P[j]) < min)

min = dist(P[i], P[j]);

**return** min;

}

**float** min(**float** x, **float** y)

{

**return** (x < y) ? x : y;

}

**float** stripClosest(Point strip[], **int** size, **float** d)

{

**float** min = d;

qsort(strip, size, **sizeof**(Point), compareY);

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

**for** (**int** j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)

**if** (dist(strip[i], strip[j]) < min)

min = dist(strip[i], strip[j]);

**return** min;

}

**float** closestUtil(Point P[], **int** n)

{

**if** (n <= 3)

**return** bruteForce(P, n);

**int** mid = n / 2;

Point midPoint = P[mid];

**float** dl = closestUtil(P, mid);

**float** dr = closestUtil(P + mid, n - mid);

**float** d = min(dl, dr);

Point strip[n];

**int** j = 0;

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

**if** (abs(P[i].x - midPoint.x) < d)

strip[j] = P[i], j++;

**return** min(d, stripClosest(strip, j, d));

}

**float** closest(Point P[], **int** n)

{

qsort(P, n, **sizeof**(Point), compareX);

**return** closestUtil(P, n);

}

**int** main()

{

Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};

**int** n = **sizeof**(P) / **sizeof**(P[0]);

cout << "The smallest distance is " << closest(P, n);

**return** 0;

}

**Huffman Decoding:**

// C++ program to encode and decode a string using

// Huffman Coding.

#include <bits/stdc++.h>

#define MAX\_TREE\_HT 256

**using** **namespace** std;

map<**char**, string> codes;

map<**char**, **int**> freq;

**struct** MinHeapNode

{

**char** data; // One of the input characters

**int** freq; // Frequency of the character

MinHeapNode \*left, \*right; // Left and right child

MinHeapNode(**char** data, **int** freq)

{

left = right = NULL;

**this**->data = data;

**this**->freq = freq;

}

};

**struct** compare

{

**bool** **operator**()(MinHeapNode \*l, MinHeapNode \*r)

{

**return** (l->freq > r->freq);

}

};

**void** printCodes(**struct** MinHeapNode \*root, string str)

{

**if** (!root)

**return**;

**if** (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

**void** storeCodes(**struct** MinHeapNode \*root, string str)

{

**if** (root == NULL)

**return**;

**if** (root->data != '$')

codes[root->data] = str;

storeCodes(root->left, str + "0");

storeCodes(root->right, str + "1");

}

priority\_queue<MinHeapNode \*, vector<MinHeapNode \*>, compare>

minHeap;

**void** HuffmanCodes(**int** size)

{

**struct** MinHeapNode \*left, \*right, \*top;

**for** (map<**char**, **int**>::iterator v = freq.begin();

v != freq.end(); v++)

minHeap.push(**new** MinHeapNode(v->first, v->second));

**while** (minHeap.size() != 1)

{

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = **new** MinHeapNode('$',

left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

storeCodes(minHeap.top(), "");

}

**void** calcFreq(string str, **int** n)

{

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

freq[str[i]]++;

}

string decode\_file(**struct** MinHeapNode \*root, string s)

{

string ans = "";

**struct** MinHeapNode \*curr = root;

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

{

**if** (s[i] == '0')

curr = curr->left;

**else**

curr = curr->right;

**if** (curr->left == NULL **and** curr->right == NULL)

{

ans += curr->data;

curr = root;

}

}

**return** ans + '\0';

}

**int** main()

{

string str = "geeksforgeeks";

string encodedString, decodedString;

calcFreq(str, str.length());

HuffmanCodes(str.length());

cout << "Character With there Frequencies:\n";

**for** (**auto** v = codes.begin(); v != codes.end(); v++)

cout << v->first << ' ' << v->second << endl;

**for** (**auto** i : str)

encodedString += codes[i];

cout << "\nEncoded Huffman data:\n"

<< encodedString << endl;

decodedString = decode\_file(minHeap.top(), encodedString);

cout << "\nDecoded Huffman Data:\n"

<< decodedString << endl;

**return** 0;

}

**Min Cost Path:**

// Min cost path

#include <bits/stdc++.h>

**using** **namespace** std;

#define row 3

#define col 3

**int** minCost(**int** cost[row][col])

{

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

cost[i][0] += cost[i - 1][0];

**for** (**int** j = 1; j < col; j++)

cost[0][j] += cost[0][j - 1];

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

**for** (**int** j = 1; j < col; j++)

cost[i][j] += min(cost[i - 1][j - 1],

min(cost[i - 1][j], cost[i][j - 1]));

**return** cost[row - 1][col - 1];

}

**int** main(**int** argc, **char** **const** \*argv[])

{

**int** cost[row][col] = {{1, 2, 3}, {4, 8, 2}, {1, 5, 3}};

cout << minCost(cost) << endl;

**return** 0;

}

**LCS:**

#include <iostream>

#include <vector>

**using** **namespace** std;

string longestCommonSubsequence(**const** string &a, **const** string &b)

{

**int** m = a.length();

**int** n = b.length();

vector<vector<**int**>> dp(2, vector<**int**>(n + 1, 0));

**for** (**int** i = m - 1; i >= 0; i--)

{

**for** (**int** j = n - 1; j >= 0; j--)

{

**if** (a[i] == b[j])

{

dp[i % 2][j] = 1 + dp[(i + 1) % 2][j + 1];

}

**else**

{

dp[i % 2][j] = max(dp[(i + 1) % 2][j], dp[i % 2][j + 1]);

}

}

}

**int** len = dp[0][0];

string lcs;

lcs.reserve(len);

**int** i = 0, j = 0;

**while** (i < m && j < n)

{

**if** (a[i] == b[j])

{

lcs.push\_back(a[i]);

i++;

j++;

}

**else** **if** (dp[(i + 1) % 2][j] >= dp[i % 2][j + 1])

{

i++;

}

**else**

{

j++;

}

}

cout << "Longest Common Subsequence: " << lcs << endl;

**return** lcs;

}

**int** main()

{

string a, b;

cout << "Enter the first string: ";

cin >> a;

cout << "Enter the second string: ";

cin >> b;

string lcs = longestCommonSubsequence(a, b);

cout << "Length of LCS: " << lcs.length() << endl;

**return** 0;

}

**Maximum Ways(Limited):**

#include <iostream>

#include <vector>

**using** **namespace** std;

**long** **long** maxWaysToMakeChange(vector<pair<**int**, **int**>> &coins, **int** amount)

{

vector<**long** **long**> dp(amount + 1, 0);

dp[0] = 1;

**for** (**const** **auto** &coin : coins)

{

vector<**long** **long**> temp(amount + 1, 0);

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

{

**for** (**int** k = 0; k <= coin.second && j + k \* coin.first <= amount; k++)

{

temp[j + k \* coin.first] += dp[j];

}

}

dp = temp;

}

**return** dp[amount];

}

**int** main()

{

vector<pair<**int**, **int**>> coins = {{1, 0}, {2, 0}, {3, 0}, {4, 0}};

**int** amount = 0;

**for** (**auto** &coin : coins)

{

cout << "Enter coin limit for " << coin.first << ": ";

cin >> coin.second;

}

cout << "Enter the amount: ";

cin >> amount;

**long** **long** ways = maxWaysToMakeChange(coins, amount);

cout << "Maximum number of ways to make change: " << ways << endl;

**return** 0;

}

**Minimum Number of Coins(Limited):**#include <iostream>

#include <vector>

#include <algorithm>

**using** **namespace** std;

**const** **int** INF = 1e9; // A large enough value to represent infinity

**int** coinChange(vector<pair<**int**, **int**>> &coins, **int** amount)

{

**int** n = coins.size();

vector<**int**> dp(amount + 1, INF);

dp[0] = 0;

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

{

**for** (**int** j = coins[i].first; j <= amount; j++)

{

**int** numCoins = min(coins[i].second, (j - coins[i].first) / coins[i].first + 1);

dp[j] = min(dp[j], dp[j - numCoins \* coins[i].first] + numCoins);

}

}

**return** (dp[amount] == INF) ? -1 : dp[amount];

}

**int** main()

{

vector<pair<**int**, **int**>> coins = {{5, 0}, {10, 0}, {20, 0}, {50, 0}, {100, 0}, {200, 0}}; // Predefined coin values

**int** amount = 0;

**for** (**auto** &coin : coins)

{

cout << "Enter coin limit for " << coin.first << ": ";

cin >> coin.second;

}

cout << "Enter the amount: ";

cin >> amount;

**int** result = coinChange(coins, amount);

cout << "Minimum number of coins: " << result << endl;

**return** 0;

}

**Min Number of Coins:**#include <bits/stdc++.h>

**using** **namespace** std;

#define ll long long int

**int** main()

{

**int** coins[5] = {2, 5, 7, 10, 20};

**int** n = 35;

**int** ans[n + 1];

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

{

ans[i] = i / coins[0];

}

**for** (**int** i = 1; i < 5; i++)

{

**for** (**int** j = coins[i]; j <= n; j++)

{

ans[j] = min(ans[j], 1 + ans[j - coins[i]]);

}

}

cout << ans[n] << endl;

**return** 0;

}

**Total Number of Ways:**#include <bits/stdc++.h>

**using** **namespace** std;

#define ull unsigned long long int

**int** main()

{

**int** n;

**while** (cin >> n)

{

**int** coin[6] = {0, 1, 5, 10, 25, 50};

vector<ull> ans(n + 1, 0);

ans[0] = 1;

**for** (**int** i = 1; i < 6; i++)

{

**for** (**int** j = coin[i]; j <= n; j++)

{

ans[j] += ans[j - coin[i]];

}

}

**if** (ans[n] == 1)

{

cout << "There is only " << 1 << " way to produce " << n << " cents change." << endl;

}

**else**

{

cout << "There are " << ans[n] << " ways to produce " << n << " cents change." << endl;

}

}

**return** 0;

}

**Count Unique Paths:**

#include <bits/stdc++.h>

**using** **namespace** std;

**int** numberOfPaths(**int** m, **int** n)

{

**int** path = 1;

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

{

path \*= i;

path /= (i - n + 1);

}

**return** path;

}

**int** main()

{

cout << numberOfPaths(3, 3);

**return** 0;

}

**Unique Paths in a Grid With Obstacles:**#include <bits/stdc++.h>

#define int long long

**using** **namespace** std;

**int** n, m;

**int** path(vector<vector<**int**>> &dp,

vector<vector<**int**>> &grid, **int** i, **int** j)

{

**if** (i < n && j < m && grid[i][j] == 1)

**return** 0;

**if** (i == n - 1 && j == m - 1)

**return** 1;

**if** (i >= n || j >= m)

**return** 0;

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

**return** dp[i][j];

**int** left = path(dp, grid, i + 1, j);

**int** right = path(dp, grid, i, j + 1);

**return** dp[i][j] = left + right;

}

**int** uniquePathsWithObstacles(vector<vector<**int**>> &grid)

{

n = grid.size();

m = grid[0].size();

**if** (n == 1 && m == 1 && grid[0][0] == 0)

**return** 1;

**if** (n == 1 && m == 1 && grid[0][0] == 1)

**return** 0;

vector<vector<**int**>> dp(n, vector<**int**>(m, -1));

path(dp, grid, 0, 0);

**if** (dp[0][0] == -1)

**return** 0;

**return** dp[0][0];

}

**signed** main()

{

vector<vector<**int**>> v{{0, 0, 0},

{0, 1, 0},

{0, 0, 0}};

cout << uniquePathsWithObstacles(v) << " \n";

**return** 0;

}

**Printing Item In 0/1 Knapsack:**

#include <bits/stdc++.h>

#include <iostream>

**using** **namespace** std;

**int** max(**int** a, **int** b) { **return** (a > b) ? a : b; }

**void** printknapSack(**int** W, **int** wt[], **int** val[], **int** n)

{

**int** i, w;

**int** K[n + 1][W + 1];

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

{

**for** (w = 0; w <= W; w++)

{

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

K[i][w] = 0;

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

K[i][w] = max(val[i - 1] +

K[i - 1][w - wt[i - 1]],

K[i - 1][w]);

**else**

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

}

}

**int** res = K[n][W];

cout << res << endl;

w = W;

**for** (i = n; i > 0 && res > 0; i--)

{

**if** (res == K[i - 1][w])

**continue**;

**else**

{

cout << " " << wt[i - 1];

res = res - val[i - 1];

w = w - wt[i - 1];

}

}

}

**int** main()

{

**int** val[] = {60, 100, 120};

**int** wt[] = {10, 20, 30};

**int** W = 50;

**int** n = **sizeof**(val) / **sizeof**(val[0]);

printknapSack(W, wt, val, n);

**return** 0;

}

**LCIS:**

#include <bits/stdc++.h>

**using** **namespace** std;

**int** LCIS(**int** arr1[], **int** n, **int** arr2[], **int** m)

{

**int** table[m];

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

table[j] = 0;

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

{

**int** current = 0;

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

{

**if** (arr1[i] == arr2[j])

**if** (current + 1 > table[j])

table[j] = current + 1;

**if** (arr1[i] > arr2[j])

**if** (table[j] > current)

current = table[j];

}

}

**int** result = 0;

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

**if** (table[i] > result)

result = table[i];

**return** result;

}

**int** main()

{

**int** arr1[] = {3, 4, 9, 1};

**int** arr2[] = {5, 3, 8, 9, 10, 2, 1};

**int** n = **sizeof**(arr1) / **sizeof**(arr1[0]);

**int** m = **sizeof**(arr2) / **sizeof**(arr2[0]);

cout << "Length of LCIS is "

<< LCIS(arr1, n, arr2, m);

**return** (0);

}

**Minimum Iterations to form a Palindrome:**

#include <bits/stdc++.h>

**using** **namespace** std;

**int** lcs(string X, string Y, **int** m, **int** n)

{

vector<**int**> prev(n + 1, 0), curr(n + 1, 0);

**int** i, j;

**for** (i = 0; i <= m; i++)

{

**for** (j = 0; j <= n; j++)

{

**if** (i == 0 || j == 0)

prev[j] = 0;

**else** **if** (X[i - 1] == Y[j - 1])

curr[j] = prev[j - 1] + 1;

**else**

curr[j] = max(prev[j], curr[j - 1]);

}

prev = curr;

}

**return** prev[n];

}

**void** reverseStr(string &str)

{

**int** n = str.length();

**for** (**int** i = 0; i < n / 2; i++)

swap(str[i], str[n - i - 1]);

}

**int** findMinInsertionsLCS(string str, **int** n)

{

string rev = "";

rev = str;

reverseStr(rev);

**return** (n - lcs(str, rev, n, n));

}

**int** main()

{

string str = "geeks";

cout << findMinInsertionsLCS(str, str.length());

**return** 0;

}

**Box Stacking:**

#include <bits/stdc++.h>

**using** **namespace** std;

**class** Box

{

**public**:

**int** length;

**int** width;

**int** height;

};

**int** dp[303];

**int** findMaxHeight(vector<Box> &boxes, **int** bottom\_box\_index, **int** index)

{

**if** (index < 0)

**return** 0;

**if** (dp[index] != -1)

**return** dp[index];

**int** maximumHeight = 0;

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

{

**if** (bottom\_box\_index == -1 || (boxes[i].length < boxes[bottom\_box\_index].length && boxes[i].width < boxes[bottom\_box\_index].width))

maximumHeight = max(maximumHeight,

findMaxHeight(boxes, i, i - 1) + boxes[i].height);

}

**return** dp[index] = maximumHeight;

}

**int** maxStackHeight(**int** height[], **int** width[], **int** length[],

**int** types)

{

vector<Box> boxes;

memset(dp, -1, **sizeof**(dp));

Box box;

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

{

box.height = height[i];

box.length = max(length[i], width[i]);

box.width = min(length[i], width[i]);

boxes.push\_back(box);

box.height = width[i];

box.length = max(length[i], height[i]);

box.width = min(length[i], height[i]);

boxes.push\_back(box);

box.height = length[i];

box.length = max(width[i], height[i]);

box.width = min(width[i], height[i]);

boxes.push\_back(box);

}

sort(boxes.begin(), boxes.end(), [](Box b1, Box b2)

{ **return** (b1.length \* b1.width) < (b2.length \* b2.width); });

**return** findMaxHeight(boxes, -1, boxes.size() - 1);

}

**int** main()

{

**int** length[] = {4, 1, 4, 10};

**int** width[] = {6, 2, 5, 12};

**int** height[] = {7, 3, 6, 32};

**int** n = **sizeof**(length) / **sizeof**(length[0]);

printf("The maximum possible height of stack is %d\n",

maxStackHeight(height, length, width, n));

**return** 0;

}

**Tile Stacking:**

#include <bits/stdc++.h>

**using** **namespace** std;

#define N 100

**int** possibleWays(**int** n, **int** m, **int** k)

{

**int** dp[N][N];

**int** presum[N][N];

memset(dp, 0, **sizeof** dp);

memset(presum, 0, **sizeof** presum);

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

{

dp[0][i] = 0;

presum[0][i] = 1;

}

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

presum[i][0] = dp[i][0] = 1;

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

{

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

{

dp[i][j] = presum[i - 1][j];

**if** (j > k)

{

dp[i][j] -= presum[i - 1][j - k - 1];

}

}

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

presum[i][j] = dp[i][j] + presum[i][j - 1];

}

**return** dp[m][n];

}

**int** main()

{

**int** n = 3, m = 3, k = 2;

cout << possibleWays(n, m, k) << endl;

**return** 0;

}

**Ways of arrange Balls:**

#include <bits/stdc++.h>

**using** **namespace** std;

#define MAX 100

**int** countUtil(**int** p, **int** q, **int** r)

{

**int** dp[MAX][MAX][MAX];

memset(dp, 0, **sizeof**(dp));

dp[1][0][0] = 1;

dp[0][1][0] = 1;

dp[0][0][1] = 1;

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

{

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

{

**for** (**int** k = 0; k <= r; k++)

{

**if** (i == 1 && j == 0 && k == 0)

**continue**;

**if** (i == 0 && j == 1 && k == 0)

**continue**;

**if** (i == 0 && j == 0 && k == 1)

**continue**;

**if** (i - 1 >= 0)

dp[i][j][k] += dp[i - 1][j][k];

**if** (j - 1 >= 0)

dp[i][j][k] += dp[i][j - 1][k];

**if** (k - 1 >= 0)

dp[i][j][k] += dp[i][j][k - 1];

}

}

}

**return** dp[p][q][r];

}

**int** main()

{

**int** p = 1, q = 1, r = 1;

printf("%d", countUtil(p, q, r));

**return** 0;

}

**Partition Problem:**

#include <bits/stdc++.h>

**using** **namespace** std;

**bool** findPartiion(**int** arr[], **int** n)

{

**int** sum = 0;

**int** i, j;

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

sum += arr[i];

**if** (sum % 2 != 0)

**return** **false**;

**bool** part[sum / 2 + 1];

**for** (i = 0; i <= sum / 2; i++)

{

part[i] = 0;

}

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

{

**for** (j = sum / 2; j >= arr[i];

j--)

{

**if** (part[j - arr[i]] == 1 || j == arr[i])

part[j] = 1;

}

}

**return** part[sum / 2];

}

**int** main()

{

**int** arr[] = {1, 3, 3, 2, 3, 2};

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

**if** (findPartiion(arr, n) == **true**)

cout << "Can be divided into two subsets of equal "

"sum";

**else**

cout << "Can not be divided into"

<< " two subsets of equal sum";

**return** 0;

}

**Assign Unique Cap to Every Person:**

// C++ program to find number of ways to wear hats

#include <bits/stdc++.h>

#define MOD 1000000007

**using** **namespace** std;

vector<**int**> capList[101];

**int** dp[1025][101];

**int** allmask;

**long** **long** **int** countWaysUtil(**int** mask, **int** i)

{

**if** (mask == allmask)

**return** 1;

**if** (i > 100)

**return** 0;

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

**return** dp[mask][i];

**long** **long** **int** ways = countWaysUtil(mask, i + 1);

**int** size = capList[i].size();

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

{

**if** (mask & (1 << capList[i][j]))

**continue**;

**else**

ways += countWaysUtil(mask | (1 << capList[i][j]), i + 1);

ways %= MOD;

}

**return** dp[mask][i] = ways;

}

**void** countWays(**int** n)

{

string temp, str;

**int** x;

getline(cin, str);

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

{

getline(cin, str);

stringstream ss(str);

**while** (ss >> temp)

{

stringstream s;

s << temp;

s >> x;

capList[x].push\_back(i);

}

}

allmask = (1 << n) - 1;

memset(dp, -1, **sizeof** dp);

cout << countWaysUtil(0, 1) << endl;

}

**int** main()

{

**int** n; // number of persons in every test case

cin >> n;

countWays(n);

**return** 0;

}

**STL:**

binary\_search(startaddress, endaddress, valuetofind)

auto it = std::lower\_bound(start, end, key);

auto it = std::upper\_bound(start, end, key);

auto it = std::find(start, end, value);

int count = std::count(start, end, value);

auto it = std::search(start, end, subsequence\_start, subsequence\_end);

std::rotate(start, middle, end);

std::merge(first1, last1, first2, last2, result);

bool sorted = std::is\_sorted(start, end);

auto max\_it = std::max\_element(start, end);

auto min\_it = std::min\_element(start, end);