//Modular Arithmatic

//(a+b)%m = ((a%m) + (b%m))%m

//(a\*b)%m = ((a%m) \* (b%m))%m

inline ull odd(ull n) {

return (n & 1); //returns true if n is odd faster than n%2

}

inline ull powmod(ull n, ull p, ull m) {

if(p == 0) return 1;

if(!odd(p)) {

ull tmp = powmod(n, p/2, m)%m;

return (tmp\*tmp)%m;

}

else return ((n%m)\*(powmod(n, p-1, m)%m))%m;

}

inline ull plusmod(ull x, ull y, ull m) {

return ((x%m)+(y%m))%m;

}

//Prime Generator and factorization

bitset<N>bit;

vector<ll> factors, prime;

ll power[N];

void sieve() {

bit.set();

bit[0] = bit[1] = 0;

for(ll i = 0; i <= N; i++) { //it can be limited to sqrt(N)

if(bit[i]) { //here it is not used as we want to0

for(ll j = i \* i; j <= N; j += i) //save the primes in prime vector

bit[j] = 0;

prime.pb(i);

} } }

void primeFactor\_of\_factorial(ll n) { //n!

memset(power, 0, sizeof(power));

for(size\_t i = 0; prime[i] <= n && i < prime.size(); i++) {

int tmp = n;

wh(tmp) {

power[prime[i]] += tmp / prime[i]; //if we want to generate powers and numbers

//factors.pb(prime[i]); //if we only to genetare all the numbers

tmp /= prime[i];

} } }

void primeFactor(ll n) {

memset(power, 0, sizeof(power));

if(prime[n]) { //First determine if n is a prime number

power[n]++;

//factors.pb(n);

}

else {

for(size\_t i = 0; prime[i]\*prime[i] <= n && i < prime.size(); i++) {

wh(n % prime[i] == 0) {

power[prime[i]]++;

//factors.pb(prime[i]);

n/=prime[i];

} }

if(n > 1) { //Must be a prime number which is not in prime[i]

power[n]++; //it would happen if n is a large number

//factors.pb(n);

} }

//Subsets (2^n)

int main() {

int len, s = 0, sub\_sum\_find, tmp, subset\_sum[10000];

scanf(" %d", &len);

int arr[len+1]; //arr is containing the numbers

for(register int i = 0; i < len; i++)

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

scanf("%d", &sub\_sum\_find);

for(register int i = 0; i < (1 << len); i++) {

subset\_sum[s] = 0;

for(register int j = 0; j < len; j++)

if(i & (1 << j)) //this point can be noted by saving i

subset\_sum[s] += arr[j];

if(subset\_sum[s] == sub\_sum\_find) tmp = i;

s++;

}

for(register int j = 0; j < len; j++)

if(tmp & (1 << j))

printf("%d ", arr[j]); //generates the numbers

return 0;

}

//2D Max Sum

int main() {

register int n, i, j, k, l, maxsubrect, subrect;

int A[110][110];

while(scanf(" %d", &n) != EOF) {

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

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

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

if(i > 0) A[i][j] += A[i-1][j];

if(j > 0) A[i][j] += A[i][j-1];

if(i > 0 && j > 0) A[i][j] -= A[i-1][j-1];

}

maxsubrect = -127\*100\*100;

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

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

for(k = i; k < n; k++)

for(l = j; l < n; l++) {

subrect = A[k][l];

if(i > 0) subrect -= A[i-1][l];

if(j > 0) subrect -= A[k][j-1];

if(i > 0 && j > 0) subrect += A[i-1][j-1];

maxsubrect = max(maxsubrect, subrect);

}

printf("2D Max Sum: %d\n", maxsubrect);

}

return 0;

}

//1D Max Sum

sum = mx = 0;

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

sum += a[i]; //a[i] contains the numbers

if(sum < 0) sum = 0;

else if(sum > mx) mx = sum;

}

pf("1D Max Sum: %lld\n", mx);

//Coin Change (DP)

// n is the amount we need to produce

// coin[] array contains the coins we can use

int coin[] = {1, 2, 3}, test[1000];

int main() {

while(1) {

int n, coin\_amount = 3;

scanf("%d", &n);

// Solution for producing amount with coins. Without any co-occurance and

// coins can be used more than once

// Bottom Up solution

memset(test, 0, sizeof(test));

test[0] = 1; // Base case

for(register int i = 0; i < coin\_amount; i++) // this will NOT produce co-occurance

for(register int j = 1; j<=n; j++) // solution for 4 if there is present 1 & 2 coins would be 3

if(j >= coin[i]) // 1+1+2, 2+2, 1+1+1+1

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

printf("Solution without co-occurance : %d\n", test[n]);

// Solution for producing amount with coins. With co-occurance and

// coins can be used more than once

// Bottom Up solution

memset(test, 0, sizeof(test));

test[0] = 1; // Base case

for(register int j = 1; j <= n; j++) // this will produce co-occurance

for(register int i = 0; i < coin\_amount; i++) // solution for 4 if there is present 1 & 2 coins would be 5

if(j >= coin[i]) // 1+1+2, 2+2, 1+1+1+1

test[j] += test[j - coin[i]]; // and also 2+1+1, 1+2+1

printf("Solution with co-occurance : %d\n", test[n]);

// Solution for producing amount with coins. With co-occurance and

// coins can be used more than once

// Top Down solution

memset(test, inf, sizeof(test));

test[0] = 0; // Base case

for(register int i = 0; i < coin\_amount; i++) // this will produce co-occurance

for(register int j = n; j > 0; j--) // solution for 4 if there is present 1, 2 & 3 coins would be 2

if(j >= coin[i] && (test[j - coin[i]] + 1) < inf) // 1+3, and 3+1

test[j] = test[j-coin[i]] + 1;

printf("Solution by using coins only once with co-occurance : %d\n", test[n]);

}

return 0;

}

//Data Structure

//Segment Tree

int arr[N], tree[4\*N]; //Always take the tree 4 times bigger

void segment\_build(int pos, int L, int R) {

tree[pos] = 0;

if(L==R) {

tree[pos] = arr[L];

return;

}

int mid = (L+R)/2;

segment\_build(pos\*2, L, mid);

segment\_build(pos\*2+1, mid+1, R);

tree[pos] = tree[pos\*2] \* tree[pos\*2+1];

}

void segment\_update(int pos, int L, int R, int i, int val) {

if(L==R) {

tree[pos] = val;

return;

}

int mid = (L+R)/2;

if(i <= mid)

segment\_update(pos\*2, L, mid, i, val);

else

segment\_update(pos\*2+1, mid+1, R, i, val);

tree[pos] = tree[pos\*2] \* tree[pos\*2+1];

}

int segment\_query(int pos, int L, int R, int l, int r) {

if(R < l || r < L) return 1;

if(l <= L && R <= r) return tree[pos];

int mid = (L+R)/2;

int x = segment\_query(pos\*2, L, mid, l, r);

int y = segment\_query(pos\*2+1, mid+1, R, l, r);

return x\*y;

}

//Data Structure

//Union Disjoint Set

ll u\_set[N+100], u\_list[N+100];

//u\_set is used to determine set

//u\_list is used to keep track of the nodes that each node connects (as a root)

inline ll root(ll n) { //finding the root of a set

if(u\_set[n] == n)

return n;

else

return u\_set[n] = root(u\_set[n]); //path compression

}

inline ll make\_union(ll a, ll b) { //make union of set, returns the value of

ll x = root(a); //the new root

ll y = root(b);

if(x == y) //returns the same value if the input two

return x; //value is same

else if(u\_list[x] > u\_list[y]) {

u\_set[y] = x;

u\_list[x] += u\_list[y];

return x;

}

else {

u\_set[x] = y;

u\_list[y] += u\_list[x];

return y;

} }

void union\_init(ll l) { //initialising of set and list

for(ll i = 0; i <= l; i++) {

u\_list[i] = 1;

u\_set[i] = i;

} }

//Graph Theory

//BFS

//Shortest Path in unweighted graph

//the level from a node u is the shortest path from u to any node in unweighted graph

void bfs(int u) {

queue<int>q; //parent[v] = u

visited[u] = 1;

level[u] = 0;

parent[u] = -1; //the source's parent is tagged

q.push(u); //pushing the starting node in queue

wh(!q.empty()) {

int U = q.front();

q.pop();

for(size\_t i = 0; i < g[U].size(); i++) { //using adjency list g[node]

int v = g[U][i];

if(!visited[v]) {

level[v] += level[u]+1; //saving the distance

past[v] = U; //the parent nodes are saved

visited[v] = 1; //visited nodes are tagged

q.push(v); //visited nodes are pushed for next //iteration

}}}}

//in main functio

for(int i = 0; i < node; i++) if(!visited[i]) bfs(i); //check every connected/non connected node

memset(visited, 0, sizeof(visited)); //to track the nodes which are visited

memset(parent, 0, sizeof(parent)); //to track the parent nodes

//BFS Bipartite

//if the graph cycle is odd then it is not bi-colorable

bool bipartite(int u) {

queue<int>q;

q.push(u);

color[u] = 0; //color must be memset to inf in main func.

isBipartite = true; //tag to check if its bipartite

while(!q.empty()) {

int U = q.top();

q.pop();

for(size\_t i = 0; i < g[U].size(); i++) {

int v = g[u][i];

if (color[v] == INF) { //instead of recording distance,

color[v] = 1 - color[u]; //just record two colors {0, 1}

q.push(v);

}

else if (color[v.first] == color[u]) { // u & v has same color

isBipartite = false;

break; //we have a coloring conflict

} } }

//DFS

//basic implimentation

//check if node v is visitable from u. if so, dfs\_num[v] == 1

void dfs(int u) {

dfs\_num[u] = 1; //dfs\_num zero initialized in main(), set counter to 1

for (size\_t i = 0; i < g[u].size(); i++) { // Adjency list

int v = g[u][j]; // v is the visitable node from node u (u -> v)

if (dfs\_num[v] == 0) // important check to avoid cycle, its not visited

dfs(v); // recursively visits unvisited neighbors of vertex u

} }

//Flood Fill

//Size of connected component

int dr[] = {-1, -1, -1, 0, 0, 1, 1, 1};

int dc[] = {-1, 0, +1, -1, +1, -1, 0, +1}; // trick to explore an implicit 2D grid

int floodfill(int r, int c) {

if(r < 0 || r >= R || c < 0 || c >= C) return 0; //checking the bounderies

if(g[r][c] != tag || visited[g[r][c]]) return 0; //checking if the grid is valid

cc\_size++; //increasing connected component size

visited[g[r][c]] = 1;

for(int i = 0; i < 8; i++)

floodfill(r + dr[i], c + dc[i]); //recursion to all other side grids

}

//Topological Sort

//Directed Acyclic Graph (DAG)

void topology(int u)

{

visited[u] = 1;

for(size\_t i = 0; i < g[u].size(); i++)

if(visited[g[u][i]] == 0)

dfs2(g[u][i]);

topsort.push(u); //its a stack

}

//DFS spanning tree / forest

//UNVISITED -> 0

//EXPLORED -> 1

//VISITED -> 2

//tree edge

//back edge (cycle)

//forward edge (cross edge)

void graphCheck(int u) {

dfs\_num[u] = 1;

for(size\_t i = 0; i < g[u].size(); i++) {

int v = g[u][i];

if(dfs\_num[v] == 0) { //if the node is unvisited, tree edge

dfs\_parent[v] = u; //the parent of v is u

graphCheck(v);

}

else if(dfs\_num[v] == 1) { //if the node is explored, but full dfs not done

if(dfs\_parent[u] == v) //if the node's parent is its child node (Undirected)

pf("Two ways (%d %d)-(%d %d)\n", u, v, v, u);

else //only option is left is backedge

pf("Back Edge (%d %d) (Cycle)\n", u, v);

}

else if(dfs\_num[v] == 2) //it the child node's dfs is done, its a forward edge

pf("Forward/Cross Edge (%d %d)\n", u, v);

}

dfs\_num[u] = 2; //in this point the full dfs is done

}