**//1D Max Sum**

**//Algorithm : Jay Kadane**

**//Complexity : O(n)**

main() {

int n;

scanf("%d", &n);

int A[n+1];

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

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

//Main part of the code

int sum = 0, ans = 0;

for(int i = 0; i < 9; i++) {

sum += A[i];

ans = max(sum, ans); //always take the larger sum

if(sum < 0)

sum = 0; //if sum is negative, reset it (greedy)

}

printf("1D Max Sum : %d\n", ans);

}

**//2D Max Sum**

**//DP, Inclusion Exclusion**

**//Complexity : O(n^4)**

int main() {

int row\_column, A[100][100]; //A square matrix

scanf("%d", &row\_column);

for(int i = 0; i < row\_column; i++) //input of the matrix/2D array

for(int j = 0; j < row\_column; j++) {

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

if(i > 0)

A[i][j] += A[i-1][j]; //take from right

if(j > 0)

A[i][j] += A[i][j-1]; //take from left

if(i > 0 && j > 0)

A[i][j] -= A[i-1][j-1]; //inclusion exclusion

}

int maxSubRect = -1e7;

for(int i = 0; i < row\_column; i++) //i & j are the starting coordinate of sub-rectangle

for(int j = 0; j < row\_column; j++)

for(int k = i; k < row\_column; k++) //k & l are the finishing coordinate of sub-rectangle

for(int l = j; l < row\_column; l++) {

int 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]; //due to inclusion exclusion

maxSubRect = max(subRect, maxSubRect);

}

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

return 0;

}

**// 0-1 Knapsack**

**// Dynamic Programming**

**//Note : val array contains element values starting from 1 index, 0 index is empty**

int Knapsack(int totalWeight, int val[], int totalElements) {

int dp[50001][101]; // DP Table [BagWeight][TotalElements]

//int track[101] = {0}; // Use this if you want to print the taken elements

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

dp[i][0] = 0; // Base Case

// Calculating best weight(that will be taken) for every possible element

for(int i = 1; i <= totalElements; i++) { // Element starts from 1

for(int weight = 1; weight <= totalWeight; weight++) {

if(val[i] > weight) // If elements weight is greater than available weight

dp[weight][i] = dp[weight][i-1]; // Skip this element

else { // If enough space is available for this element **\***

if(dp[weight][i-1] >= dp[weight-val[i]][i-1] + val[i])

dp[weight][i] = dp[weight][i-1]; // If ignoring this element causes good outcome, ignore this

else {

dp[weight][i] = dp[weight - val[i]][i-1] + val[i]; // Otherwise take this element

//track[dp[weight][i]] = i; // This tracks the taken element

} } } }

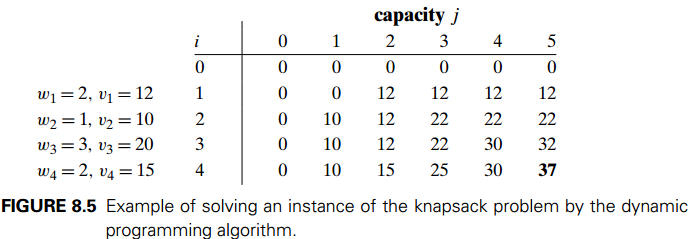
**/\* These code outputs the taken values**

int sumWeight = dp[totalWeight][totalElements];

int lastTakenElement = track[sumWeight];

while(lastTakenElement != 0) { // lastTakenElement is the index of element

printf("%d ", val[lastTakenElement]); // val[lastTakenElement] is taken value

 sumWeight -= val[lastTakenElement];

lastTakenElement = track[sumWeight];

}

printf("\n"); **\*/**

return dp[totalWeight][totalElements];

}

// 0-1 knapsack top down method

// Left side code runs from higher limit to zero, right side (commented code) runs from 0 to higher limit

// Left side code contains element values in array starting from index 1, in right side code element values in array starts from 0

int Knapsack(int weight, int i) { // int Knapsack(int weight, int i) {

if(i == 0 || weight == 0) // if( i == element\_number) {

return 0; // if(weight > weight\_limit)

if(dp[weight][i] != -1) // return -INF; // Returning an inf makes this impossible

return dp[weight][i]; // return 0; }

if(elementWeight[i] > weight) // if(dp[weight][i] != -1)

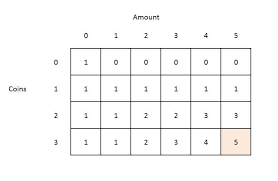
return dp[weight][i] = Knapsack(weight, i-1); // return dp[weight][i];

else // return dp[i][w] = max(Knapsack(weight, i-1), Knapsack(weight – elementWeight[i], i-1) + cost[i]);

return dp[i][w] = max(Knapsack(weight, i-1), Knapsack(weight – elementWeight[i], i-1) + cost[i]);

}

**// Coin Change**

**// All Possible Types**

int main() {

int n, coin\_amount = 3; // n = value to produce

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

// 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 Fig: Coin Change Table

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

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(int j = 1; j <= n; j++) // This will produce co-occurrence

for(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-occurrence : %d\n", test[n]);

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

// Coins can be used more than once

// Bottom up solution

for(int i = 0; i <= 1000; i++)

test[i] = inf; // Normal case

test[0] = 0; // Base case

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

for(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-occurrence : %d\n", test[n]);

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

// coins can be used more than once

// Bottom up solution

for(int i = 0; i <= 1000; i++)

test[i] = inf; // Normal case

test[0] = 0; // Base case

for(register int i = n; i > 0; i--) // this will NOT produce co-occurrence

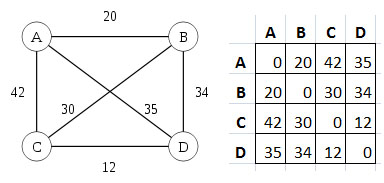
for(register int j = 0; j < coin\_amount; j++) // solution for 4 if there is present 1, 2 & 3 coins would be 1

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

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

printf("Solution by using coins only once without co-occurrence : %d\n", test[n]);

return 0;

}

**// Traveling Salesman**

**// Time Complexity : O(2^n \* n^2)**

//dist[u][v] = distance from u to v

//dp[u][bitmask] = dp[node][set\_of\_taken\_nodes] (saves the best(min/max) path)

//call with tsp(starting node, 1)

// Best solution may be more than one

Fig: Traveling Salesman Problem State (A-B-C-D-A)

int n, x[11], y[11], dist[11][11], memo[11][1 << 11], dp[11][1 << 11]; //This example is for 11 routes/nodes

int tsp(int u, int bitmask) { // Starting node and bitmask of taken nodes

if(bitmask == ((1 << (1+n)) - 1)) // When it steps in this node, if all nodes are visited

return dist[u][0]; // Then return to 0'th (starting) node [as the path is **Hamiltonian**]

//or use return dist[u][start] if starting node is not 0

if(dp[u][bitmask] != -1) // If we have previous value set up

return dp[u][bitmask]; // Use that previous value

int ans = 1e9; // Set an infinite value

for(int v = 0; v <= n; v++) // For all the nodes

if(u != v && !(bitmask & (1 << v))) // if this node is not the same node, and if this node is not used // yet(in bitmask)

ans = min(ans, dist[u][v] + tsp(v, bitmask | (1 << v)));

//min(past\_val, dist u->v + dist(v->all other untaken nodes))

return dp[u][bitmask] = ans; //save in dp and return

}

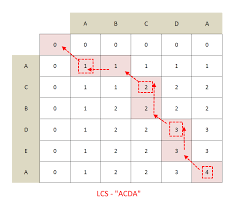
**//Longest Common Sub Sequence**

**//Dynamic Programming**

char a[210], b[210];

int dp[210][210], len\_a, len\_b;

//LCS is the same sequence in two strings: a s x z and s x z a. Here LCS is 3 {a, sx, z}

int LCS(char a[], char b[], int len\_a, int len\_b) {

dp[210][210] = 0;

for(register int i = 1; i <= len\_a; i++)

for(register int j = 1; j <= len\_b; j++) {

if(i == 0 || j == 0) //base case

dp[i][j] = 0;

else if(a[i-1] == b[j-1]) //if a match found

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

else Fig: Longest Common Subsequence

dp[i][j] = max(dp[i-1][j], dp[i][j-1]); // dp[i][j] = max(ignoring b[j-1] (taking b[j]), ignoring a[i-1] (taking a[i]))

}

return dp[len\_a][len\_b];

}