

//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][1];
    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) {
        printf("%d ", val[lastTakenElement]);
        sumWeight -= val[lastTakenElement];
        lastTakenElement = track[sumWeight];
    }
    printf("\n"); */

    return dp[totalWeight][totalElements];
}

```

		capacity j						
		i	0	1	2	3	4	5
		0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12	
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22	
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32	
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37	

FIGURE 8.5 Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

// 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

// Index of elementWeight[] and cost[] starts from 1

```
int Knapsack(int weight, int i) {
    if(i == 0 || weight == 0)
        return 0;
    if(dp[weight][i] != -1)
        return dp[weight][i];
    if(elementWeight[i] > weight)
        return dp[weight][i] = Knapsack(weight, i-1);
    else
        return dp[i][w] = max(Knapsack(weight, i-1), Knapsack(weight - elementWeight[i], i-1) + cost[i]);
}
```

// Runs Knapsack from limit to 0

// If weigh is zero or nothing is taken

// If weight if this element is more than available

// Ignore the element

// Skip, Take

// Index of elementWeight[] and cost[] starts from 0

```
int Knapsack(int weight, int i) {
    if( i == element_number) {
        if(weight > weight_limit)
            return -INF;
        return 0; }
    if(dp[weight][i] != -1)
        return dp[weight][i];
    return dp[i][w] = max(Knapsack(weight, i-1), Knapsack(weight - elementWeight[i], i-1) + cost[i]);
}
```

// Runs Knapsack from 0 to limit

// If weight is somehow more than the limit

// Returning an INF so that, this value is ignored

// Coin Change

// All Possible Types

```
int main() {
    int n, coin_amount = 3;
    int coin[] = {1, 2, 3}, test[1000];
```

// n = value to produce

// coin[] = coin values

// Solution for producing amount with coins. Without any co-occurrence 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++)

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

if(j >= coin[i])

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

printf("Solution without 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

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

test[0] = 1;

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

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

if(j >= coin[i])

// This will NOT produce co-occurrence

// Solution for 4 if there is present 1 & 2 coins would be 3

// 1+1+2, 2+2, 1+1+1+1

// Base case

// This will produce co-occurrence

// Solution for 4 if there is present 1 & 2 coins would be 5

// 1+1+2, 2+2, 1+1+1+1

		Amount					
		0	1	2	3	4	5
Coins	0	1	0	0	0	0	0
	1	1	1	1	1	1	1
	2	1	1	2	2	3	3
	3	1	1	2	3	4	5

Fig: Coin Change Table

```

    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

```

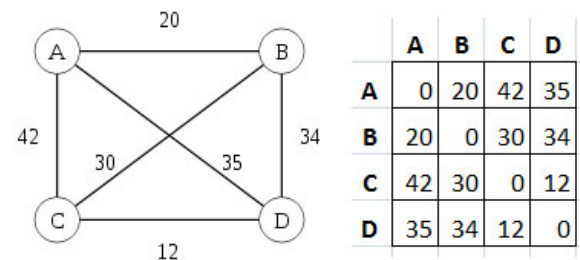


Fig: Traveling Salesman Problem State (A-B-C-D-A)

```

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) {
    if(bitmask == ((1 << (1+n)) - 1)) // Starting node and bitmask of taken nodes
        return dist[u][0];           // When it steps in this node, if all nodes are visited
    // 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;
for(int v = 0; v <= n; v++)
    if(u != v && !(bitmask & (1 << v)))
        // Set an infinite value
        // For all the nodes
        // 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
```

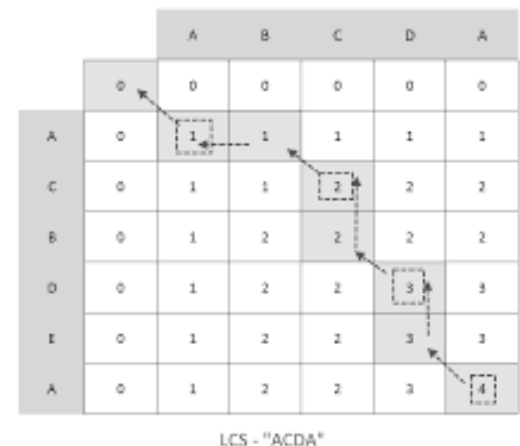
```
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]); // dp[i][j] = max(ignoring
```

```
                b[j-1] (taking b[j]), ignoring a[i-1] (taking a[i]))
```

```
            }
```

```
    return dp[len_a][len_b];
```

```
}
```



// Longest Increasing Subsequence

// Time Complexity : n log n

```
int LIS(vector<int> &val) {
```

```
    vector<int> lis;
```

```
    for(int i = 0; i < (int)val.size(); i++) {
```

```
        vector<int>::iterator it = lower_bound(lis.begin(), lis.end(), val[i]);
```

```
        if(lis.end() == it)
```

```
            lis.push_back(val[i]);
```

```
        else
```

```
            lis[it - lis.begin()] = val[i];
```

```
    }
```

```
    return lis.size();
```

```
}
```

// In case if some tweak needed in binary search, use this instead

```
int lowerBound(std::vector<int>&v, int low, int hi, int key) {
```

```
    while(hi - low > 1) {
```

```
        int m = (low+hi)/2;
```

// Less than-equal to the key

// If low <= hi was used

```
    if(v[m] >= key)
        hi = m;
    else
        low = m;
}
return hi;
}
```

// hi = m - 1
// low = m + 1

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

Dynamic Solve, **Complexity : n^2**

// Use this solve for **path printing**

```
int lis(int val[], int size) {
    int lis[size+10] = {1};

    for(int i = 1; i < size; ++i)
        for(int j = 0; j < i; ++j)
            if(val[i] > val[j] && lis[i] < lis[j]+1)
                lis[i] = lis[j] + 1;

    return max_element(lis, lis+size);
}
```

// Default LIS value
// This is the finish point
// For all past points
// If finish point is higher than past point and taking it
// optimizes the result, then take it
// Find the maximum element

Fig : LIS Dynamic Solve