Dynamic Programming

Some standard Dynamic Programming solutions with different complexities are given here. By doing all these types anyone can easily solve most of the DP problems.
(Do upvote if you find this useful.)

0/1 Knapsack (Bounded)

```
Problem Statement- Given weights and values of n items, put
these items in a knapsack of capacity W to get the maximum
total value in the knapsack.
```

```
int knapsack(int wt[], int val[], int n, int w) {
    if(n == 0 or w == 0) return 0;
    if(dp[n][w] != -1) return dp[n][w];

    if(wt[n-1] > w)
        return dp[n][w] = knapsack(wt, val, n-1, w);

    return dp[n][w] = max(knapsack(wt, val, n-1, w-wt[n-1]) +
val[n-1], knapsack(wt, val, n-1, w));
}
int knapsack(int wt[], int val[], int n, int w) {
    int dp[n+1][w+1];
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= w; j++){
            if(wt[i-1] > j)
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = max(dp[i-1][j-wt[i-1]] + val[i-1],
dp[i-1][j]);
        }
    }
    return dp[n][w];
}
int knapsack(int wt[], int val[], int n, int w) {
    int dp[w+1];
    memset(dp, 0, sizeof(dp));
    for(int i = 0; i < n; i++) {
```

```
        for(int j = w; j >= wt[i]; j--){
            dp[j] = max(dp[j], dp[j-wt[i]] + val[i]);
        }
    }
    return dp[w];
}
```

Similar Problems-

1. Subset sum
2. Equal sum partition
3. Count of subsets sum with a given sum
4. Minimum subset sum difference
5. Count the number of subset with a given difference
6. Target sum

0/1 Knapsack (Unbounded)

```
Problem Statement- Given a rod of length w inches and an
array of prices that includes prices of pieces of size
smaller than w. Determine the maximum value obtainable by
cutting up the rod and selling the pieces.
```

```
int knapsack(int wt[], int val[], int n, int w) {
    if(n == 0 or w == 0) return 0;
    if(dp[n][w] != -1) return dp[n][w];

    if(wt[n-1] > w)
        return dp[n][w] = knapsack(wt, val, n-1, w);

    return dp[n][w] = max(knapsack(wt, val, n, w-wt[n-1]) +
val[n-1], knapsack(wt, val, n-1, w));
}
int knapsack(int wt[], int val[], int n, int w) {
    int dp[n+1][w+1];
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= w; j++){
            if(wt[i-1] > j)
                dp[i][j] = dp[i-1][j];
            else
```

```
                    dp[i][j] = max(dp[i][j-wt[i-1]] + val[i-1], 
dp[i-1][j]);
          }
     }
     return dp[n][w];
}
int knapsack(int wt[], int val[], int n, int w) {
     int dp[w+1];
     memset(dp, 0, sizeof(dp));
     for(int i = 0; i < n; i++) {
          for(int j = wt[i]; j <= w; j++){
               dp[j] = max(dp[j], dp[j-wt[i]] + val[i]);
          }
     }
     return dp[w];
}
```

Similar Problems-

1. Integer Break
2. Coin Change
3. Coin Change 2
4. Combination Sum IV
5. Perfect Squares

Longest Common Subsequence

```
Problem Statement- Given two sequences, find the length of 
longest subsequence present in both of them.
```

```
int lcs(string a, string b, int m, int n) {
     if(m == 0 or n == 0) return 0;
     if(dp[m][n] != -1) return dp[m][n];

     if(a[m-1] == b[n-1])
          return dp[m][n] = 1 + lcs(a, b, m-1, n-1);
     else
          dp[m][n] = max(lcs(a, b, m-1, n), lcs(a, b, m, n-1));
}
int lcs(string a, string b, int m, int n) {
     int dp[m+1][n+1];
     memset(dp, 0, sizeof(dp));
```

```
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            if(a[i-1] == b[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m][n];
}
int lcs(string a, string b, int m, int n) {
    int dp[2][n+1];
    memset(dp, 0, sizeof(dp));
    for(int i = 1; i <= m; ++i){
        for(int j = 1; j <= n; j++){
            if(a[i-1] == b[j-1])
                dp[i&1][j] = dp[i&1^1][j-1] + 1;
            else
                dp[i&1][j] = max(dp[i&1^1][j], dp[i&1][j-1]);
        }
    }
    return dp[m&1][n];
}
```

```
Solve it in 1-D DP
```

Similar Problems-

1. Longest common substring
2. Shortest common supersequence
3. Minimum number of insertion and deletion to convert A to B
4. Longest repeating subsequence
5. Length of longest subsequence of A which is substring of B
6. Subsequence pattern matching
7. Count how many times A appears as subsequence in B
8. Longest palindromic subsequence
9. Count of palindromic substrings
10. Minimum number of deletion in a string to make it palindrome
11. Minimum number of insertion in a string to make it palindrome

Matrix Chain Multiplication

```
Problem Statement- Given a sequence of matrices, find the
most efficient way to multiply these matrices together. The
```

```
efficient way is the one that involves the least number of
multiplications.
```

```
int mcm(int ar[], int i, int j) {
    if(i >= j) return 0;
    if(dp[i][j] != -1) return dp[i][j];
    int mn = INT_MAX;
    for(int k = i; k < j; k++) {
        int temp = mcm(ar, i, k) + mcm(ar, k+1, j) + (ar[i-1]
* ar[k] * ar[j]);
        mn = min(mn, temp);
    }
    return dp[i][j] = mn;
}
// call mcm(ar, 1, n-1);
int mcm(int ar[], int n) {
    int dp[n][n];
    memset(dp, 0, sizeof(dp));

    for(int i = n-2; i > 0; i--) {
        for(int j = i+1; j < n; j++) {
            int ans = INT_MAX;
            for(int k = i; k < j; k++)
                ans = min(ans, dp[i][k] + dp[k+1][j] +
ar[i-1] * ar[k] * ar[j]);
            dp[i][j] = dp[j][i] = ans;
        }
    }
    return dp[1][n-1];
}
```

Similar Problems-

1. Burst Balloons
2. Evaluate expression to true / boolean parenthesization
3. Minimum or maximum value of a expression
4. Palindrome partitioning
5. Scramble string
6. Super Egg Drop

Fibonacci

```
Problem Statement- Given an integer array representing the
amount of money in the houses, return the maximum amount of
money that a thief can rob, the only constraint stopping that
has to follow is that the thief cannot rob two adjacent
houses.
```

```
int rob(int ar[], int n) {
    if(n == 0) return 0;
    if(n == 1) return ar[0];
    if(dp[n] != -1) return dp[n];
    dp[n] = max(rob(ar, n-1), rob(ar, n-2) + ar[n-1]);
    return dp[n];
}
int rob(int ar[], int n) {
    int dp[n+1];
    dp[0] = 0;
    dp[1] = ar[0];
    for(int i=2; i<=n; i++){
        dp[i] = max(dp[i-1], dp[i-2]+ar[i-1]);
    }
    return dp[n];
}
// further space optimization possible (using variables)
```

Similar Problems-

1. Fibonacci number
2. Climbing stairs
3. Minimum jumps to reach the end
4. Friends pairing problem
5. Maximum subsequence sum such that no three are consecutive

Longest Increasing Subsequence

```
Problem Statement- Given an integer array, return the length
of the longest strictly increasing subsequence.
```

```
int lis(int ar[], int n) {
    if(n == 1) return dp[n] = 1;
```

```
    if(dp[n] != -1) return dp[n];

    int cur, ans = 1;
    for(int i = 1; i < n; i++) {
        cur = lis(ar, i);
        if(ar[i-1] < ar[n-1] and cur + 1 > ans)
            ans = cur + 1;
    }
    return dp[n] = ans;
}
// final answer *max_element(dp, dp+n+1);
int lis(int ar[], int n) {
    int dp[n];
    for(int i = 0; i < n; i++) dp[i] = 1;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < i; j++) {
            if(ar[i] > ar[j])
                dp[i] = max(dp[i], dp[j]+1);
        }
    }
    return *max_element(dp, dp + n);
}
```

```
Note: This problem can be solved in O(NlogN) using Greedy +
BS.
```

Similar Problems-

1. Print longest increasing subsequence
2. Number of longest increasing subsequences
3. Longest non-decreasing subsequence
4. Find the longest increasing subsequence in circular manner
5. Longest bitonic subsequence
6. Longest arithmetic subsequence
7. Maximum sum increasing subsequence

DP on Trees

```
Problem Statement 1- Given a tree T of N nodes, where each
node i has Ci coins attached with it. You have to choose a
subset of nodes such that no two adjacent nodes(i.e. nodes
```

```
connected directly by an edge) are chosen and sum of coins
attached with nodes in chosen subset is maximum.
```

```
vector<int> adj[N]; // adjacency list
int dp1[N]; // when including node V
int dp2[N]; // when not including node V
int C[N]; // coins

void dfs(int u, int p) {
    int sum1 = 0;
    int sum2 = 0;

    for(auto v: adj[u]) {
        if(v == p) continue;
        dfs(v, u);
        sum1 += dp2[v];
        sum2 += max(dp1[v], dp2[v]);
    }
    dp1[u] = C[u] + sum1;
    dp2[u] = sum2;
}
// let tree is rooted at 1
// then ans = max(dp1[1], dp2[1]);
```

```
Problem Statement 2- Given a tree T of N nodes, calculate
longest path between any two nodes(also known as diameter of
tree).
```

```
vector<int> adj[N]; // adjacency list
int f[N]; // when longest path starts from node x and goes
into its subtree
int g[N]; // when longest path starts in subtree of x, passes
through x and ends in subtree of x
int diameter; // final diameter

void dfs(int u, int p) {
    // this vector will store f for all children of u
    vector<int> fvals;
    for(auto v: adj[u]) {
        if(v == p) continue;
        dfs(v, u);
        fvals.push_back(f[v]);
    }
```

```
    //sort to get top two values
    //we can also get top two values without sorting(think
about it) in O(n)
    //current complexity is O(n log n)
    sort(fvals.begin(), fvals.end(), greater<int>());
    f[u] = 1;
    if(!fvals.empty()) f[u] += fvals[0];
    if(fvals.size() >= 2) g[u] = 1 + fvals[0] + fvals[1];

    diameter = max(diameter, max(f[u], g[u]));
}
```

Similar Problems-

1. Diameter of Binary Tree
2. Binary Tree Maximum Path Sum
3. Unique Binary Search Trees II
4. House Robber III

DP on Grids

```
Problem Statement- Given a m x n grid filled with non-
negative numbers, find a path from top left to bottom right,
which minimizes the sum of all numbers along its path. You
can only move either down or right at any point in time.
```

```
int minCost(vector<vector<int>>grid, int m, int n){
    if(m == 0 or n == 0) return INT_MAX;
    if(m == 1 and n == 1) return grid[0][0];
    if(dp[m][n] != -1) return dp[m][n];

    dp[m][n] = grid[m-1][n-1] + min(minCost(grid, m-1, n),
minCost(grid, m, n-1));

    return dp[m][n];
}
// minCost(grid, m, n);
int minCost(vector<vector<int>>grid, int m, int n){
    int dp[m][n];
    dp[0][0] = grid[0][0];
    for (int i = 1; i < m; ++i)
        dp[i][0] = dp[i - 1][0] + grid[i][0];
```

```
    for (int j = 1; j < n; ++j)
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    for (int i = 1; i < m; ++i)
        for (int j = 1; j < n; ++j)
            dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j
- 1]);
    return dp[m - 1][n - 1];
}
```

Similar Problems-

1. Unique Paths
2. Unique Paths II
3. Minimum Path Sum
4. Dungeon Game
5. Cherry Pickup

Digit DP

```
Problem Statement- How many numbers x are there in the range
a to b, where the digit d occurs exactly k times in x?
```

```
#include <bits/stdc++.h>
using namespace std;

vector<int> num;
int a, b, d, k;
int DP[12][12][2];
// DP[p][c][f] = Number of valid numbers <= b from this state
// p = current position from left side (zero based)
// c = number of times we have placed the digit d so far
// f = the number we are building has already become smaller
than b? [0 = no, 1 = yes]

int helper(int pos, int cnt, int f){
    if(cnt > k) return 0;

    if(pos == num.size()){
        if(cnt == k) return 1;
        return 0;
    }
```

```
    if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
    int res = 0;

    int LMT;

    if(f == 0){
        // Digits we placed so far matches with the prefix of
b
        // So if we place any digit > num[pos] in the current
position, then the number will become greater than b
        LMT = num[pos];
    } else {
        // The number has already become smaller than b. We
can place any digit now.
        LMT = 9;
    }

    // Try to place all the valid digits such that the number
doesn't exceed b
    for(int dgt = 0; dgt<=LMT; dgt++){
        int nf = f;
        int ncnt = cnt;
        if(f == 0 && dgt < LMT) nf = 1; // The number is
getting smaller at this position
        if(dgt == d) ncnt++;
        if(ncnt <= k) res += helper(pos+1, ncnt, nf);
    }

    return DP[pos][cnt][f] = res;
}

int solve(int b){
    num.clear();
    while(b>0){
        num.push_back(b%10);
        b/=10;
    }
    reverse(num.begin(), num.end());
    // Stored all the digits of b in num for simplicity

    memset(DP, -1, sizeof(DP));
    int res = helper(0, 0, 0);
    return res;
}

int main () {
    cin >> a >> b >> d >> k;
    int res = solve(b) - solve(a-1);
```

```
    // we can also use 4th state as to check number is greater than or eual to a so that we don't have to recur twice
    cout << res << endl;

    return 0;
}
```

Similar Problems-

1. Number of Digit One
2. Non-negative Integers without Consecutive Ones
3. Numbers At Most N Given Digit Set
4. Numbers With Repeated Digits
5. Number of integers having sum divisible by k

DP + Bitmask

This trick is usually used when one of the variables have very small constraints that can allow exponential solutions.

```
Problem Statement- There are n (1 <= n <= 10) people and 40 types of hats labeled from 1 to 40. Given a list of list of integers hats, where hats[i] is a list of all hats preferred by the i-th person. Return the number of ways that the n people wear different hats to each other. Since the answer may be too large, return it modulo 10^9 + 7.
```

```
const int MOD = 1e9+7;
// Bitmask on n
// maximum value of mask can be 1<<10 all over the cases
int dp[1<<10][41];
// as we are iterating over caps so we have to store that which cap can be choosen by whom
vector<int> caps[41];
int done; // maximum bitmask for a given n

int helper(int mask, int hat) {
    if(mask == done) return 1;
```

```
    // when no hat is left
    if(hat > 40) return 0;

    if(dp[mask][hat] != -1) return dp[mask][hat];
    // number of ways when given hat is not choosen by any 
person
    int cnt = helper(mask, hat+1);
    cnt %= MOD;
    int n = caps[hat].size();

    // iterating over all the persons who can choose the 
given hat
    for(int i=0; i<n; i++) {
        // continue when the person already have a hat
        if(mask & (1<<caps[hat][i])) continue;
        // set the mask true for person when he worn a hat
        cnt += helper(mask | (1<<caps[hat][i]), hat+1);
        cnt %= MOD;
    }
    return dp[mask][hat] = cnt;
}

int numberWays(vector<vector<int>>& hats) {
    int n = hats.size();
    memset(dp, -1, sizeof dp);
    done = (1<<n)-1;
    for(int i=0; i<n; i++) {
        for(auto hat: hats[i]) {
            caps[hat].push_back(i);
        }
    }
    return helper(0, 1);
}
```

Similar Problems-

1. Travelling salesman problem
2. Find minimum sum Hamiltonian Path
3. Task allotment to minimise the cost
4. Maximum Students Taking Exam
5. Find the Shortest Superstring
6. Minimum Number of Work Sessions to Finish the Tasks
7. Number of Ways to Wear Different Hats to Each Other