# Formula, formula and formula

One reason why DP problem is difficult, because you need to spend time to find the formula and build up the DP calculation. This makes DP more like a math problem instead of a typical CS problem. I personally do not like to have such problem in interview, but as you know, chanllenge is daily life for a programmer.

## 123. Best Time to Buy and Sell Stock III

Hard

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

**Note:**You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

  Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

**Example 2:**

**Input:** [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

  Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are

  engaging multiple transactions at the same time. You must sell before buying again.

**Example 3:**

**Input:** [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transaction is done, i.e. max profit = 0.

### Analysis:

First assume that we have no money, so buy1 means that we have to borrow money from others, we want to borrow less so that we have to make our balance as max as we can(because this is negative).

sell1 means we decide to sell the stock, after selling it we have price[i] money and we have to give back the money we owed, so we have price[i] - |buy1| = prices[i ] + buy1, we want to make this max.

buy2 means we want to buy another stock, we already have sell1 money, so after buying stock2 we have buy2 = sell1 - price[i] money left, we want more money left, so we make it max

sell2 means we want to sell stock2, we can have price[i] money after selling it, and we have buy2 money left before, so sell2 = buy2 + prices[i], we make this max.

So sell2 is the most money we can have.

/// <summary>

/// Leet code #123. Best Time to Buy and Sell Stock III

///

/// Say you have an array for which the ith element is the price of a given

/// stock on day i.

/// Design an algorithm to find the maximum profit. You may complete at most

/// two transactions.

/// Note: You may not engage in multiple transactions at the same time

/// (i.e., you must sell the stock before you buy again).

///

/// Example 1:

/// Input: [3,3,5,0,0,3,1,4]

/// Output: 6

/// Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3),

/// profit = 3-0 = 3. Then buy on day 7 (price = 1) and sell on

/// day 8 (price = 4), profit = 4-1 = 3.

///

/// Example 2:

/// Input: [1,2,3,4,5]

/// Output: 4

/// Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),

/// profit = 5-1 = 4.

///

/// Note that you cannot buy on day 1, buy on day 2 and sell them later,

/// as you are engaging multiple transactions at the same time. You must

/// sell before buying again.

///

/// Example 3:

/// Input: [7,6,4,3,1]

/// Output: 0

/// Explanation: In this case, no transaction is done, i.e. max profit = 0.

/// </summary>

int LeetCode::maxProfitTwoTxns(vector<int>& prices)

{

if (prices.size() == 0)

{

return 0;

}

vector<int> balance(prices.size());

for (size\_t s = 0; s < 2; s++)

{

int best\_buy = INT\_MIN;

int best\_profit = 0;

for (size\_t i = 0; i < prices.size(); i++)

{

best\_buy = max(best\_buy, balance[i] - prices[i]);

best\_profit = max(best\_profit, best\_buy + prices[i]);

balance[i] = best\_profit;

}

}

return balance[balance.size() - 1];

}

## 629. K Inverse Pairs Array

Hard

Given two integers n and k, find how many different arrays consist of numbers from 1 to n such that there are exactly k inverse pairs.

We define an inverse pair as following: For ith and jth element in the array, if i < j and a[i] > a[j] then it's an inverse pair; Otherwise, it's not.

Since the answer may be very large, the answer should be modulo 109 + 7.

**Example 1:**

**Input:** n = 3, k = 0

**Output:** 1

**Explanation:**

Only the array [1,2,3] which consists of numbers from 1 to 3 has exactly 0 inverse pair.

**Example 2:**

**Input:** n = 3, k = 1

**Output:** 2

**Explanation:**

The array [1,3,2] and [2,1,3] have exactly 1 inverse pair.

**Note:**

1. The integer n is in the range [1, 1000] and k is in the range [0, 1000].

### Analysis:

p[n][k] denotes the number of arrays that have k inverse pairs for array composed of 1 to n  
we can establish the recursive relationship between dp[n][k] and dp[n-1][i]:

if we put n as the last number then all the k inverse pair should come from the first n-1 numbers  
if we put n as the second last number then there's 1 inverse pair involves n so the rest k-1 comes from the first n-1 numbers  
...  
if we put n as the first number then there's n-1 inverse pairs involve n so the rest k-(n-1) comes from the first n-1 numbers

dp[n][k] = dp[n-1][k]+dp[n-1][k-1]+dp[n-1][k-2]+...+dp[n-1][k+1-n+1]+dp[n-1][k-n+1]

It's possible that some where in the right hand side the second array index become negative, since we cannot generate negative inverse pairs we just treat them as 0, but still leave the item there as a place holder.

dp[n][k] = dp[n-1][k]+dp[n-1][k-1]+dp[n-1][k-2]+...+dp[n-1][k+1-n+1]+dp[n-1][k-n+1]  
dp[n][k+1] = dp[n-1][k+1]+dp[n-1][k]+dp[n-1][k-1]+dp[n-1][k-2]+...+dp[n-1][k+1-n+1]

so by deducting the first line from the second line, we have

dp[n][k+1] = dp[n][k]+dp[n-1][k+1]-dp[n-1][k+1-n]

/// <summary>

/// Leet code #629. K Inverse Pairs Array

///

/// Given two integers n and k, find how many different arrays consist of

/// numbers from 1 to n such that there are exactly k inverse pairs.

/// We define an inverse pair as following: For ith and jth element in the

/// array, if i < j and a[i] > a[j] then it's an inverse pair;

/// Otherwise, it's not.

/// Since the answer may very large, the answer should be modulo 10^9 + 7.

/// Example 1:

/// Input: n = 3, k = 0

/// Output: 1

/// Explanation:

/// Only the array [1,2,3] which consists of numbers from 1 to 3 has

/// exactly 0 inverse pair.

///

/// Example 2:

/// Input: n = 3, k = 1

/// Output: 2

/// Explanation:

/// The array [1,3,2] and [2,1,3] have exactly 1 inverse pair.

///

/// Note:

/// The integer n is in the range [1, 1000] and k is in the range

/// [0, 1000].

/// </summary>

int LeetCode::kInversePairs(int n, int k)

{

unsigned long long M = 1000000007;

vector<vector<long long>> dp = vector<vector<long long>>(n + 1, vector<long long>(k + 1));

dp[0][0] = 1;

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

{

dp[i][0] = 1;

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

{

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

if (j - i >= 0)

{

// It must + mod, If you don't know why,

// you can check the case 1000, 1000

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

}

}

}

return (int)dp[n][k];

}

## 920. Number of Music Playlists

Hard

Your music player contains N different songs and she wants to listen to L(not necessarily different) songs during your trip.  You create a playlist so that:

* Every song is played at least once
* A song can only be played again only if K other songs have been played

Return the number of possible playlists.  **As the answer can be very large, return it modulo 10^9 + 7**.

**Example 1:**

**Input:** N = 3, L = 3, K = 1

**Output:** 6

**Explanation**: There are 6 possible playlists. [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

**Example 2:**

**Input:** N = 2, L = 3, K = 0

**Output:** 6

**Explanation**: There are 6 possible playlists. [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2, 1], [2, 1, 2], [1, 2, 2]

**Example 3:**

**Input:** N = 2, L = 3, K = 1

**Output:** 2

**Explanation**: There are 2 possible playlists. [1, 2, 1], [2, 1, 2]

**Note:**

1. 0 <= K < N <= L <= 100

### Analysis:

dp[i][j] denotes the solution of i songs with j different songs. So the final answer should be dp[L][N]

Think one step before the last one, there are only cases for the answer of dp[i][j]  
case 1 (the last added one is new song): listen i - 1 songs with j - 1 different songs, then the last one is definitely new song with the choices of N - (j - 1).  
Case 2 (the last added one is old song): listen i - 1 songs with j different songs, then the last one is definitely old song with the choices of j  
if without the constraint of K, the status equation will be  
dp[i][j] = dp[i-1][j-1] \* (N - (j-1)) + dp[i-1][j] \* j

If with the constaint of K, there are also two cases  
Case 1: no changes since the last added one is new song. Hence, there is no conflict  
Case 2: now we don't have choices of j for the last added old song. It should be updated j - k because k songs can't be chosed from j - 1 to j - k. However, if j <= K, this case will be 0 because only after choosing K different other songs, old song can be chosen.

if (j > k)  
dp[i][j] = dp[i-1][j-1] \* (N- (j-1)) + dp[i-1][j] \* (j-k)  
else  
dp[i][j] = dp[i-1][j-1] \* (N- (j-1))

/// <summary>

/// Leet code #920. Number of Music Playlists

///

/// Your music player contains N different songs and she wants to listen to L

/// (not necessarily different) songs during your trip. You create a playlist

/// so that:

///

/// Every song is played at least once

/// A song can only be played again only if K other songs have been played

/// Return the number of possible playlists. As the answer can be very large,

/// return it modulo 10^9 + 7.

///

///

/// Example 1:

/// Input: N = 3, L = 3, K = 1

/// Output: 6

/// Explanation: There are 6 possible playlists. [1, 2, 3], [1, 3, 2],

/// [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

///

/// Example 2:

/// Input: N = 2, L = 3, K = 0

/// Output: 6

/// Explanation: There are 6 possible playlists. [1, 1, 2], [1, 2, 1],

/// [2, 1, 1], [2, 2, 1], [2, 1, 2], [1, 2, 2]

///

/// Example 3:

/// Input: N = 2, L = 3, K = 1

/// Output: 2

/// Explanation: There are 2 possible playlists. [1, 2, 1], [2, 1, 2]

///

/// Note:

/// 1. 0 <= K < N <= L <= 100

/// </summary>

int LeetCodeDP::numMusicPlaylists(int N, int L, int K)

{

size\_t M = 1000000007;

vector<vector<long long>> dp(L + 1, vector<long long>(N + 1));

dp[0][0] = 1;

for (int i = 1; i <= L; i++)

{

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

{

dp[i][j] += dp[i - 1][j - 1] \* j % M;

dp[i][j] += dp[i - 1][j] \* max(j-K, 0) % M;

}

}

int result = (int)dp[L][N];

return result;

}

## 887. Super Egg Drop

Hard

You are given K eggs, and you have access to a building with N floors from 1 to N.

Each egg is identical in function, and if an egg breaks, you cannot drop it again.

You know that there exists a floor F with 0 <= F <= N such that any egg dropped at a floor higher than F will break, and any egg dropped at or below floor F will not break.

Each *move*, you may take an egg (if you have an unbroken one) and drop it from any floor X (with 1 <= X <= N).

Your goal is to know **with certainty** what the value of F is.

What is the minimum number of moves that you need to know with certainty what F is, regardless of the initial value of F?

**Example 1:**

**Input:** K = 1, N = 2

**Output:** 2

**Explanation:**

Drop the egg from floor 1. If it breaks, we know with certainty that F = 0.

Otherwise, drop the egg from floor 2. If it breaks, we know with certainty that F = 1.

If it didn't break, then we know with certainty F = 2.

Hence, we needed 2 moves in the worst case to know what F is with certainty.

**Example 2:**

**Input:** K = 2, N = 6

**Output:** 3

**Example 3:**

**Input:** K = 3, N = 14

**Output:** 4

**Note:**

1. 1 <= K <= 100
2. 1 <= N <= 10000

### Analysis:

dp[M][K]means that, given K eggs and M moves,  
what is the maximum number of floor that we can check.

The dp equation is:  
dp[m][k] = dp[m - 1][k - 1] + dp[m - 1][k] + 1,  
which means we take 1 move to a floor,  
if egg breaks, then we can check dp[m - 1][k - 1] floors.  
if egg doesn't breaks, then we can check dp[m - 1][k - 1] floors.

dp[m][k] is similar to the number of combinations and it increase exponentially to N

**Time Complexity**:  
For time, O(NK) decalre the space, O(KlogN) running,  
For space, O(NK).

/// <summary>

/// Leet code #887. Super Egg Drop

///

/// You are given K eggs, and you have access to a building with N floors from

/// 1 to N.

///

/// Each egg is identical in function, and if an egg breaks, you cannot drop

/// it again.

///

/// You know that there exists a floor F with 0 <= F <= N such that any egg

/// dropped at a floor higher than F will break, and any egg dropped at or

/// below floor F will not break.

///

/// Each move, you may take an egg (if you have an unbroken one) and drop it

/// from any floor X (with 1 <= X <= N).

///

/// Your goal is to know with certainty what the value of F is.

///

/// What is the minimum number of moves that you need to know with certainty

/// what F is, regardless of the initial value of F?

///

/// Example 1:

///

/// Input: K = 1, N = 2

/// Output: 2

/// Explanation:

/// Drop the egg from floor 1. If it breaks, we know with certainty that

/// F = 0.

/// Otherwise, drop the egg from floor 2. If it breaks, we know with

/// certainty that F = 1.

/// If it didn't break, then we know with certainty F = 2.

/// Hence, we needed 2 moves in the worst case to know what F is with

/// certainty.

///

/// Example 2:

/// Input: K = 2, N = 6

/// Output: 3

///

/// Example 3

/// Input: K = 3, N = 14

/// Output: 4

/// Note:

/// 1 <= K <= 100

/// 1 <= N <= 10000

/// </summary>

int LeetCode::superEggDrop(int K, int N)

{

vector<vector<int>> dp;

int result = 0;

while (true)

{

dp.push\_back(vector<int>(K));

for (int k = 0; k < K; k ++)

{

if (k == 0) dp[result][k] = result + 1;

else if (result == 0)

{

dp[result][k] = 1;

}

else

{

dp[result][k] = 1 + dp[result - 1][k] + dp[result - 1][k - 1];

}

if (dp[result][k] >= N) return result + 1;

}

result++;

}

}

## 552. Student Attendance Record II

Hard

Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7.

A student attendance record is a string that only contains the following three characters:

1. **'A'** : Absent.
2. **'L'** : Late.
3. **'P'** : Present.

A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.

**Example 1:**

**Input:** n = 2

**Output:** 8

**Explanation:**

There are 8 records with length 2 will be regarded as rewardable:

"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"

Only "AA" won't be regarded as rewardable owing to more than one absent times.

**Note:** The value of **n** won't exceed 100,000.

### Analysis:

Let AnLn denote number of strings containing n A's and ending with n L's  
For example:

When n = 1 we have:

A0L0: P

A0L1: L

A0L2:

A1L0: A

A1L1:

A1L2:

Done:

When n = 2 we have:

A0L0: LP, PP

A0L1: PL

A0L2: LL

A1L0: AP, LA, PA

A1L1: AL

A1L2:

Done: AA

In general we have this transition table:

-----+---------------

INIT | A -- L -- P --

-----+---------------

A0L0 | A1L0 A0L1 A0L0

A0L1 | A1L0 A0L2 A0L0

A0L2 | A1L0 Done A0L0

A1L0 | Done A1L1 A1L0

A1L1 | Done A1L2 A1L0

A1L2 | Done Done A1L0

From the transition table we see that:  
A0L0 of n can result from A0L0 + A0L1 + A0L2 of n - 1 by appending P  
A0L1 of n can only result from A0L0 of n - 1 by appending L  
and so on...

That's why in each iteration we update:  
dp[0] = dp[0] + dp[1] + dp[2]  
dp[1] = dp[0]  
and so on...

/// <summary>

/// Leet code #552. Student Attendance Record II

///

/// Given a positive integer n, return the number of all possible

/// attendance records with length n, which will be regarded as

/// rewardable. The answer may be very large, return it after

/// mod 10^9 + 7.

/// A student attendance record is a string that only contains the

/// following three characters:

///

/// 'A' : Absent.

/// 'L' : Late.

/// 'P' : Present.

/// A record is regarded as rewardable if it doesn't contain more

/// than one 'A' (absent) or more than two continuous 'L' (late).

/// Example 1:

/// Input: n = 2

/// Output: 8

/// Explanation:

/// There are 8 records with length 2 will be regarded as rewardable:

/// "PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"

/// Only "AA" won't be regarded as rewardable owing to more than one

/// absent times.

/// Note: The value of n won't exceed 100,000.

/// </summary>

int LeetCode::checkRecord(int n)

{

unsigned long long M = 1000000007;

vector<unsigned long long> P(n), L(n), A(n);

// process for record 'P' and 'L', consider at position

// i we have 'P' or 'L', totally how many record we

// can get

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

{

if (i == 0)

{

P[i] = 1; L[i] = 1;

}

else

{

// for 'P' at i position, simply add count for 'P' and 'L'

// at position of i - 1

P[i] = P[i - 1] + L[i - 1];

// for 'P' at i position, s add count for 'P' and 'L'

// at position of i - 1, but should deduct 'LL'

L[i] = P[i - 1] + L[i - 1];

// count double LL in the lower significance and deduct it

if (i == 2) L[i] -= 1;

else if (i > 2)

{

if (L[i] >= P[i - 3]) L[i] -= P[i - 3];

else L[i] += M - P[i - 3];

}

}

P[i] %= M;

L[i] %= M;

}

// Now process for 'A' at i position

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

{

if (i == 0) A[i] = 1;

// for 'A' at i position, simply add count for 'P' and 'L'

// at position of i - 1

else A[i] = P[i - 1] + L[i - 1];

A[i] %= M;

}

// Now add record 'A' and recalculate

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

{

if (i == 0)

{

P[i] = 1; L[i] = 1;

}

else

{

// for 'P' at i position, add count for 'P', 'L', 'A'

// at position of i - 1

P[i] = P[i - 1] + L[i - 1] + A[i - 1];

// for 'P' at i position, s add count for 'P' and 'L'

// at position of i - 1, but should deduct 'LL'

L[i] = P[i - 1] + L[i - 1] + A[i - 1];

// count double LL in the lower significance and deduct it

if (i == 2) L[i] -= 1;

else if (i > 2)

{

if (L[i] >= P[i - 3] + A[i - 3])

{

L[i] -= (P[i - 3] + A[i - 3]);

}

else

{

L[i] += M - (P[i - 3] + A[i - 3]);

}

}

}

P[i] %= M;

L[i] %= M;

}

return (int)((P[n - 1] + L[n - 1] + A[n - 1]) % M);

}