<https://leetcode.com/problems/k-inverse-pairs-array/description/>

For an integer array nums, an inverse pair is a pair of integers [i, j] where 0 <= i < j < nums.length and nums[i] > nums[j].

Given two integers n and k, return the number of different arrays consisting of numbers from 1 to n such that there are exactly k inverse pairs. Since the answer can be huge, return it 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 pairs.

**Example 2:**

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

**Output:** 2

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

**Constraints:**

1 <= n <= 1000

0 <= k <= 1000

**Attempt 1: 2023-02-12**

**Solution 1: Native DFS (60 min, TLE 11/81)**

**The Brute Force solution idea brought from L46. Permutations, it is natrual to think of generating all permutations and count the inverse pairs for each permuation.**

**Wrong Solution**

**The problem happen on 'curNum', its not sync with index 'i' or value 'nums[i]', during the recursion it cannot properly removing the element in 'permutation' during the backtrack process**

class Solution {

    public int kInversePairs(int n, int k) {

        return helper(1, n, k, new ArrayList<Integer>());

    }

    private int helper(int curNum, int n, int k, List<Integer> permutation) {

        if(permutation.size() == n) {

            return isKInversePair(k, new ArrayList<>(permutation));

        }

        int count = 0;

        for(int i = 0; i < n; i++) {

            if(permutation.contains(curNum)) {

                continue;

            }

            permutation.add(curNum);

            // The problem happen on 'curNum', its not sync with index 'i' or

            // value 'nums[i]', during the recursion during the recursion it

// cannot properly removing the element in 'permutation' during

// the backtrack process

            count += helper(curNum + 1, n, k, permutation);

            permutation.remove(permutation.size() - 1);

        }

        return count;

    }

    private int isKInversePair(int k, List<Integer> permutation) {

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

            for(int j = i + 1; j < permutation.size(); j++) {

                if(permutation.get(i) > permutation.get(j)) {

                    k--;

                }

            }

        }

        return k == 0 ? 1 : 0;

    }

}

**Correct Solution**

**Fully based on L46. Permutations template and emoving 'curNum'**

class Solution {

    public int kInversePairs(int n, int k) {

        int[] nums = IntStream.range(1, n + 1).toArray();

        return helper(nums, k, new ArrayList<Integer>());

    }

    private int helper(int[] nums, int k, List<Integer> permutation) {

        // For each permutation we build by nums[1, n] check if

        // it satisfies 'k inverse pair'

        if(permutation.size() == nums.length) {

            return isKInversePair(k, new ArrayList<>(permutation));

        }

        int count = 0;

        for(int i = 0; i < nums.length; i++) {

            if(permutation.contains(nums[i])) {

                continue;

            }

            permutation.add(nums[i]);

            count += helper(nums, k, permutation);

            permutation.remove(permutation.size() - 1);

        }

        return count;

    }

    private int isKInversePair(int k, List<Integer> permutation) {

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

            for(int j = i + 1; j < permutation.size(); j++) {

                if(permutation.get(i) > permutation.get(j)) {

                    k--;

                }

            }

        }

        return k == 0 ? 1 : 0;

    }

}

=====================================================================

// We don't really need 'nums', just use 'n' is enough

class Solution {

    public int kInversePairs(int n, int k) {

        // We don't really need 'nums', just use 'n' is enough

        //int[] nums = IntStream.range(1, n + 1).toArray();

        //return helper(nums, k, new ArrayList<Integer>());

        return helper(n, k, new ArrayList<Integer>());

    }

    private int helper(int n, int k, List<Integer> permutation) {

        // For each permutation we build by nums[1, n] check if

        // it satisfies 'k inverse pair'

        //if(permutation.size() == nums.length) {

        if(permutation.size() == n) {

            return isKInversePair(k, new ArrayList<>(permutation));

        }

        int count = 0;

        //for(int i = 0; i < nums.length; i++) {

        for(int i = 0; i < n; i++) {

            //if(permutation.contains(nums[i])) {

            if(permutation.contains(i + 1)) {

                continue;

            }

            //permutation.add(nums[i]);

            permutation.add(i + 1);

            //count += helper(nums, k, permutation);

            count += helper(n, k, permutation);

            permutation.remove(permutation.size() - 1);

        }

        return count;

    }

    private int isKInversePair(int k, List<Integer> permutation) {

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

            for(int j = i + 1; j < permutation.size(); j++) {

                if(permutation.get(i) > permutation.get(j)) {

                    k--;

                }

            }

        }

        return k == 0 ? 1 : 0;

    }

}

Time Complexity: O(n^2\*n!), n^2 for isKInversePair() method, and n! for recursion

because the approach that adds n to the permutations of 1 to n - 1

**Refer to**

<https://leetcode.com/problems/k-inverse-pairs-array/solutions/678320/evolve-from-brute-force-to-dp/>

It is natrual to think of generating all permutations and count the inverse pairs for each permuation. I summarize some solutoins of 46. Permutations. If you are familiar with anyone, you can at least have a brute force solution.

Brute force, generate all permutations. O(n^2\*n!), I choose the approach that adds n to the permutations of 1 to n-1 because it is easy to compute inverse pairs.

public int kInversePairs(int n, int k) {

return dfs(1,n,k,new ArrayList<>());

}

private int dfs(int num, int n, int k, List<Integer> lst) {

if(num==n+1) return isKinv(k, lst);

int count = 0;

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

List<Integer> perm = new ArrayList<>(lst);

perm.add(i,num);

count+=dfs(num+1,n,k,perm);

}

return count;

}

private int isKinv(int k, List<Integer> lst) {

int s = lst.size();

for(int i=0;i<s;i++) for(int j=i+1;j<s;j++) if(lst.get(i)>lst.get(j)) k--;

return k==0 ? 1 : 0;

}

**Refer to**

[L46.P11.3.Permutations](note://0506470DCA494FCCB9BC6CC154718ED0)

**Brute Force style "Remove redundant 'index' from parameter" below:**

Remove redundant 'index' from parameter

class Solution {

    public List<List<Integer>> permute(int[] nums) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        // No need 'index = 0' as recursion parameter anymore, since

        // when recursion happens in "permutation", it always have to

        // start with first element to scan the whole input again, so

        // the for loop in recursion always start with i = 0, no need

        // receive value passed in from parameter 'index = 0'

        // That's the major difference than "combination" which requires

        // next recursion level always move 1 index ahead of current index

        //helper(nums, result, new ArrayList<Integer>(), 0);

        helper(nums, result, new ArrayList<Integer>());

        return result;

    }

    //private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, int index) {

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp) {

        if(tmp.size() == nums.length) {

            result.add(new ArrayList<Integer>(tmp));

            return;

        }

        //for(int i = index; i < nums.length; i++) {

        for(int i = 0; i < nums.length; i++) {

            // No boolean[] visited array required just use ArrayList.contains() to

            // identify if current number been used or not before is because one condition:

            // All the integers of nums are unique.

            if(tmp.contains(nums[i])) {

                continue;

            }

            tmp.add(nums[i]);

            // Differ than L77.Combinations statement which pass local variable 'i'

            // plus 1('i + 1') into next recursion

            //helper(nums, result, tmp, index);

            helper(nums, result, tmp);

            tmp.remove(tmp.size() - 1);

        }

    }

}

**Solution 2: Better Native DFS (120 min, TLE 34/81)**

class Solution {

    public int kInversePairs(int n, int k) {

        return helper(1, n, k);

    }

    private int helper(int curNum, int n, int k) {

        if(curNum == n + 1) {

            return k == 0 ? 1 : 0;

        }

        int count = 0;

        for(int i = Math.max(0, k - (curNum - 1)); i <= k; i++) {

            // Inserting 'curNum' may create at most 'curNum - 1' inversions,

            // so the remaining inversions is from max(0, k - (curNum - 1)) to k

            count += helper(curNum + 1, n, i);

        }

        return count;

    }

}

Time Complexity: O(n!), n! for recursion because the approach that adds n to the permutations of 1 to n - 1

**Step by step**

When adding n to the permutation of 1 to n - 1, the numbers on the right of n are all newly created inverse pairs with n. So we can compute the inverse pairs when generating the permutations.

==============================================================================

e.g: n = 3, k = 1

We will iterately add num from 1 to n(=3)

(1) num = 1, it will have at most 0 inverse pair (because we will only have at most num - 1 inverse pairs, for now is 1 - 1 = 0), current permutation is {1}

------------------------------------------------------------------------------

(2) num = 2, it will have at most 1 inverse pair (because we will only have at most num - 1 inverse pairs, for now is 2 - 1 = 1), at most 1 inverse pairs not equal k = 1 requirement:

a. 1,2 create a combination has no inverse pair -> {1,2}, which add 2 after 1, no inverse pair

b. 1,2 create a combination has one inverse pair(match k = 1 requirement) -> {2,1}, which add 2 before 1, {2,1} is the inverse pair

------------------------------------------------------------------------------

(3) num = 3, it will have at most 2 inverse pairs (because we will only have at most num - 1 inverse pairs, for now is 3 - 1 = 2), at most 2 inverse pairs not equal k = 1 requirement, we have to obey k = 1:

a. 1,2,3 create a combination has no inverse pair -> {1,2,3}, which add 3 after {1,2}, no inverse pair, abandon

b. 1,2,3 create a combination has one inverse pair -> {1,3,2}, which add 3 between {1,2}, {3,2} is the inverse pair, match k = 1 inverse pair

c. 1,2,3 create a combination has one inverse pair -> {2,1,3}, which add 3 after {2,1}, {2,1} is the inverse pair

d. 1,2,3 create a combination has two inverse pair -> {3,1,2}, which add 3 before {1,2}, {3,1} and {3,2} are inverse pairs, same for {3,2,1},{2,3,1}, not equal requirement k = 1 (max inverse pair), abandon

------------------------------------------------------------------------------

after all, for n = 3, k = 1, we have {1,3,2} and {2,1,3} two combinations satisfy k = 1 inverse pair

**Refer to**

<https://leetcode.com/problems/k-inverse-pairs-array/solutions/678320/evolve-from-brute-force-to-dp/>

Better brute force O(n!) When adding n to the permutation of 1 to n - 1, the numbers on the right of n are all newly created inverse pairs with n. So we can compute the inverse pairs when generating the permutations. We no longer need the actual permutation becasue we can compute inverse pair without it.

public int kInversePairs(int n, int k) {

    return dfs(1, n, k);

}

private int dfs(int num, int n, int k) {

    if(num == n + 1) {

        return k == 0 ? 1 : 0;

    }

    int count = 0;

    for(int i = Math.max(0, k - num + 1); i <= k; i++) {

        // inserting 'num' may create at most 'num - 1' inversions, so the

        // remaining inversions is from max(0, k - (num - 1)) to k

        count += dfs(num + 1, n, i);

    }

    return count;

}

**Solution 3: Better Native DFS + Memoization (10 min, TLE 81/81, test cases passed but took too long)**

class Solution {

    int MOD = (int)1e9 + 7;

    public int kInversePairs(int n, int k) {

        Integer[][] memo = new Integer[n + 1][k + 1];

        return helper(1, n, k, memo);

    }

    private int helper(int curNum, int n, int k, Integer[][] memo) {

        if(curNum == n + 1) {

            return k == 0 ? 1 : 0;

        }

        if(memo[curNum][k] != null) {

            return memo[curNum][k];

        }

        int count = 0;

        for(int i = Math.max(0, k - (curNum - 1)); i <= k; i++) {

            // Inserting 'curNum' may create at most 'curNum - 1' inversions,

            // so the remaining inversions is from max(0, k - (curNum - 1)) to k

            count = (count + helper(curNum + 1, n, i, memo)) % MOD;

        }

        return memo[curNum][k] = count;

    }

}

Time Complexity: O(k\*n^2), k\*n for go through on all memo status during recursion,

another n for on each recursion level has a for loop

**Refer to**

<https://leetcode.com/problems/k-inverse-pairs-array/solutions/678320/evolve-from-brute-force-to-dp/>

Memoization O(k\*n^2)

public int kInversePairs(int n, int k) {

Integer[][] mem = new Integer[n+1][k+1];

return dfs(1,n,k,mem);

}

private int dfs(int num, int n, int k, Integer[][] mem) {

if(num==n+1) return k==0?1:0;

if(mem[num][k]!=null) return mem[num][k];

int count=0;

for(int i=Math.max(0,k-num+1);i<=k;i++) count= (count+dfs(num+1,n,i,mem))%1000000007;

return mem[num][k]=count;

}

**Solution 4: Bottom Up DP (30 min)**

**Important: Don't miss the inner for loop 'j' from 0 to k, if no inner for loop, 'k' will be a fixed value, but this problem solution based on a typical 2D DP, the outer for loop 'curNum' only represents 1st dimension, we need a inner for loop 'j' to represent 2nd dimension**

class Solution {

    public int kInversePairs(int n, int k) {

        int MOD = (int)1e9 + 7;

        // Top = {1, k}, bottom = {n + 1, 0}

        // dp[i][j] means number of ways to add i until n to the array with j new inversions.

        int[][] dp = new int[n + 2][k + 1];

        dp[n + 1][0] = 1;

        // Important: Don't miss the inner for loop 'j' from 0 to k, if

        // no inner for loop, 'k' will be a fixed value, but this problem

        // solution based on a typical 2D DP, the outer for loop 'curNum' only

        // represents 1st dimension, we need a inner for loop 'j' to represent

        // 2nd dimension

        // Convert recursion into nested for loop as outer for loop 'curNum'

        // from n to 1, inner for loop 'j' from 0 to k

        for(int curNum = n; curNum >= 1; curNum--) {

            for(int j = 0; j <= k; j++) {

                // Inserting 'curNum' may create at most 'curNum - 1' inversions,

                // so the remaining inversions is from max(0, j - (curNum - 1)) to j

                for(int i = Math.max(0, j - (curNum - 1)); i <= j; i++) {

                    dp[curNum][j] = (dp[curNum][j] + dp[curNum + 1][i]) % MOD;

                }

            }

        }

        return dp[1][k];

    }

}

Time Complexity: O(k\*n^2), same as Native DFS + Memoization solution, k\*n for go through on all 2D DP

status during nested for loop, another n for most inner for loop

**Refer to**

<https://leetcode.com/problems/k-inverse-pairs-array/solutions/678320/evolve-from-brute-force-to-dp/>

Bottom up DP O(k\*n^2), transform from Native DFS to DP literally.

dp[i][j] means number of ways to add i until n to the array with j new inversions.

dp[i][j]=dp[i+1][j]+dp[i+1][j-1]+...+dp[i+1][max(0,j-i+1)]

public int kInversePairs(int n, int k) {

// dp[i][j] means number of ways to add i until n to the array with j new inversions

int[][] dp = new int[n+2][k+1];

dp[n+1][0]=1;

for(int i=n;i>0;i--)

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

for(int m=Math.max(0,j-i+1);m<=j;m++)

dp[i][j] = (dp[i][j]+dp[i+1][m])%1000000007;

return dp[1][k];

}

Or we can write as a clearer description for dp[i][j].

**dp[i][j] means the number of arrays containing 1 to i and j inversions.**

dp[i][j] = dp[i-1][j]+dp[i-1][j-1]+...+dp[i-1][max(0,j-i+1)]

public int kInversePairs(int n, int k) {

// dp[i][j] means the number of arrays containing 1 to i and j inversions

int[][] dp = new int[n+1][k+1];

dp[0][0]=1;

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

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

for(int m=Math.max(0,j-i+1);m<=j;m++)

dp[i][j] = (dp[i][j]+dp[i-1][m])%1000000007;

return dp[n][k];

}

**Solution 5: DP O(nk) solution (180 min)**

class Solution {

    public int kInversePairs(int n, int k) {

        int MOD = (int)1e9 + 7;

        // dp[n][k] denotes the number of arrays that have k

        // inverse pairs for array composed of 1 to n

        int[][] dp = new int[n + 1][k + 1];

        // Base case: {0} has 0 inverse pair, 1 array found

        dp[0][0] = 1;

        // (1) dp[n][k] = dp[n - 1][k] + dp[n - 1][k - 1] + dp[n - 1][k - 2] + ...

        //            + dp[n - 1][k - (n - 1)]

        // k -> k + 1, we create (2) from (1)

        // (2) dp[n][k + 1] = dp[n - 1][k + 1] + dp[n - 1][k] + dp[n - 1][k - 1] + ...

        //                + dp[n - 1][k + 1 - (n - 1)]

        // We do (2) - (1)

        // -> dp[n][k + 1] = dp[n][k] + dp[n - 1][k + 1] - dp[n - 1][k - (n - 1)]

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

        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]) % MOD;

                if(j >= i) {

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

                    // Test out by:

                    // Input: n = 1000, k = 1000

                    // Expected: 663677020, Output: -273404233

                    dp[i][j] = (dp[i][j] + MOD) % MOD;

                }

            }

        }

        return dp[n][k];

    }

}

Time Complexity: O(n\*k)

**Refer to**

<https://leetcode.com/problems/k-inverse-pairs-array/solutions/104815/java-dp-o-nk-solution/>

**dp[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]

Below is the java code:

public static int kInversePairs(int n, int k) {

int mod = 1000000007;

if (k > n\*(n-1)/2 || k < 0) return 0;

if (k == 0 || k == n\*(n-1)/2) return 1;

long[][] dp = new long[n+1][k+1];

dp[2][0] = 1;

dp[2][1] = 1;

for (int i = 3; i <= n; i++) {

dp[i][0] = 1;

for (int j = 1; j <= Math.min(k, i\*(i-1)/2); j++) {

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

if (j >= i) dp[i][j] -= dp[i-1][j-i];

dp[i][j] = (dp[i][j]+mod) % mod;

}

}

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

}