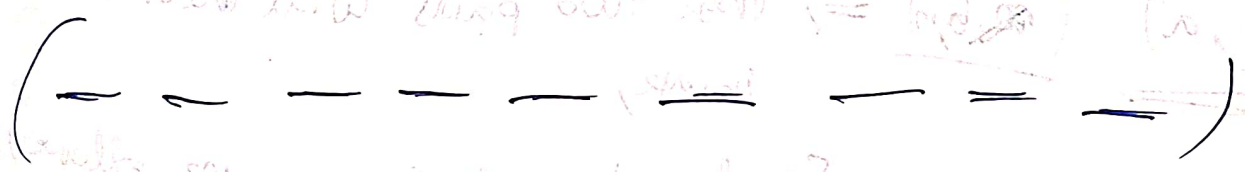
(K inverse Pairs)

Upon seeing this problem, first thing that came into my mind is bit free, idk why, seeing the inverse thing may be

But as per the requirement of (n, k)

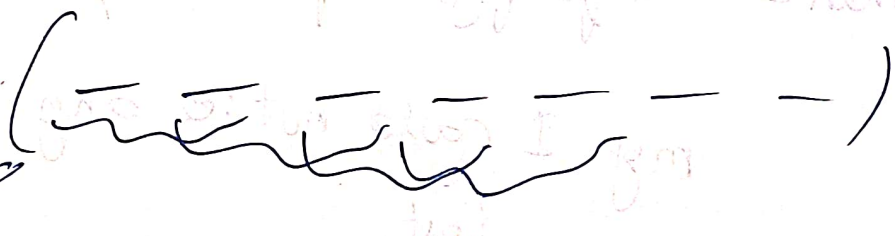
Given $n \rightarrow [1, 2, 3, \ldots, n]$ how many number of permutations will give K inverse pairs

My thought process initially was



I was thinking like how many ways

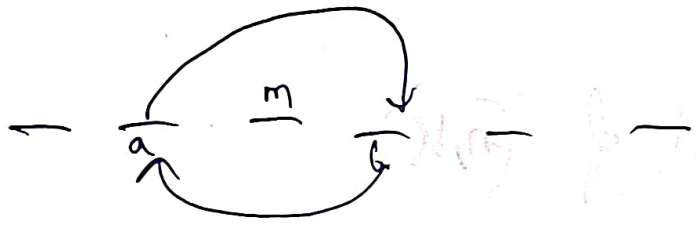
K=1 → (One inverse pair)



Given a sorted array only adjacent reversals will result into (K=1) pair inverse.

Then came for (K=2),

For (K=1) I thought only adjacent reversals will lead to K=1 shift because



Lets say in the sorted array we made a switch of

[a,b] $a < m < b$, after ~~see~~ swap.

(b, m, a) will be the order.

(b, a) (~~a~~ b, m) ⇒ these two pairs will become the inverse,

So only adjacent swaps are allowed

Then I tried for figure for K=2, K=3,

my I could not see any pattern and was lost,

Lastly I saw NeetCode's explanation.

Beautiful Step by Step intuition building it was.

(Neetcodes explanation)

Lets take the example given in code,

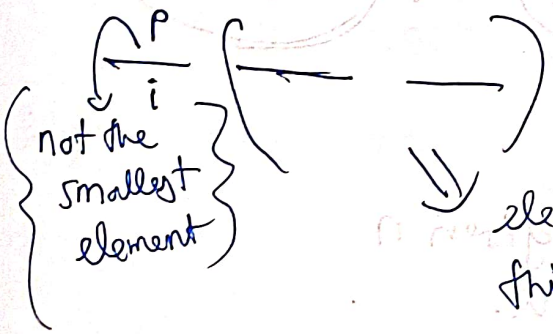
arr → (1, 2, 3) (n = 3, k = 0)

_ _ _ → 3 slots are to be filled.

what will you fill in the first slot, so that k = 0,

if in the remaining set of numbers → [_____],

if you populate anything other than the smaller number, than an inversion will get introduced.



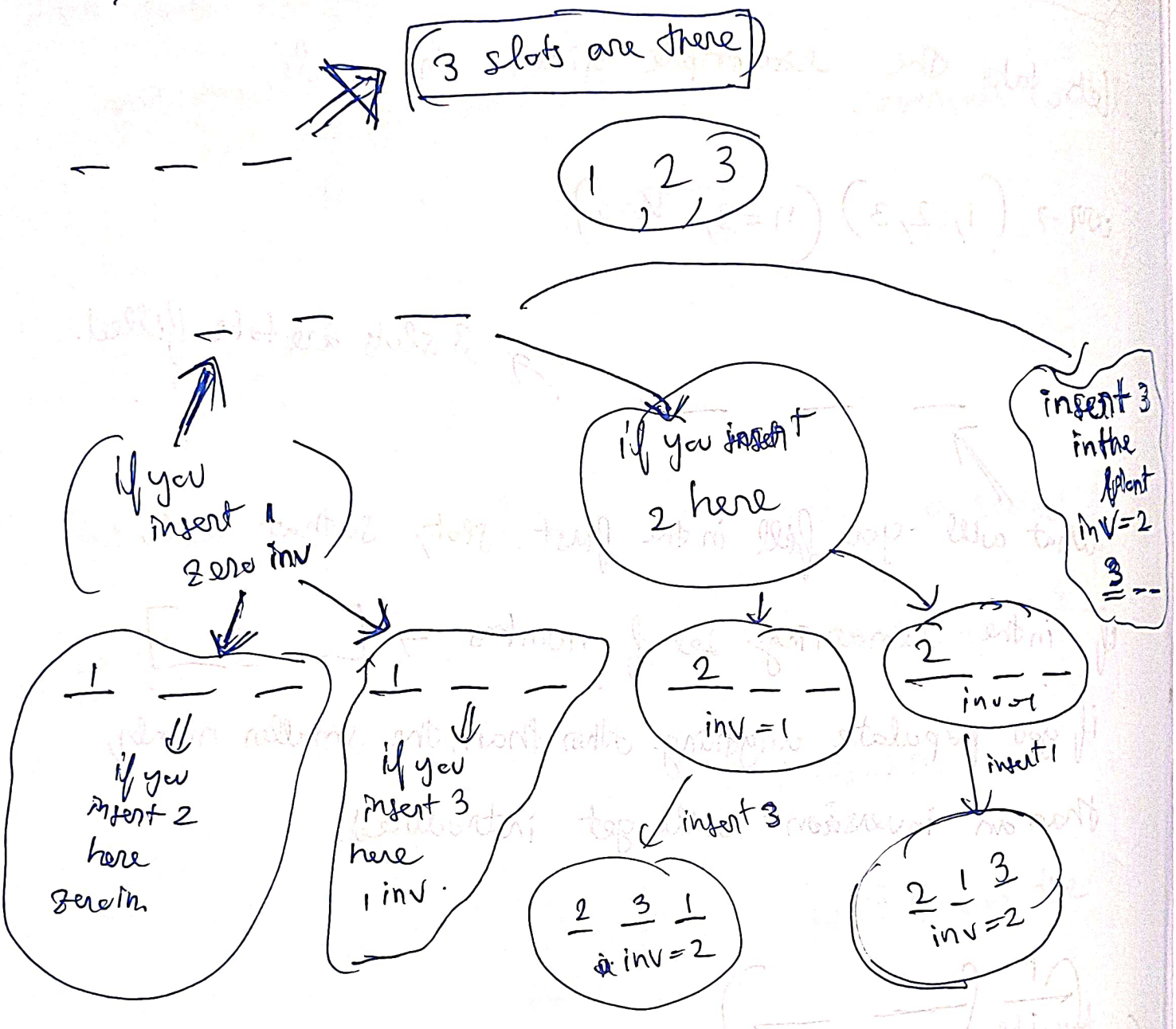
elements smaller than p, will come in this part,

So w.r.t ith index inversions introduced because of it are (p-1) elements.

(then 2 here) (then 3 here)

(Here you have to place 1).

$n=3, k=2$ $[1,2,3]$



So for a given n and k.

Imagine elements are sorted, if you place the pth element here

(p-1) inversions introduced

remaining (n-1) elements

What elements are there does not matter.

what matters is the number of elements.

For example, lets say we picked.

[1 2 3 (4) ---- n]

4th element

Remaining → [1 2 3 5 ---- n]

We have to place these in the remaining gaps.

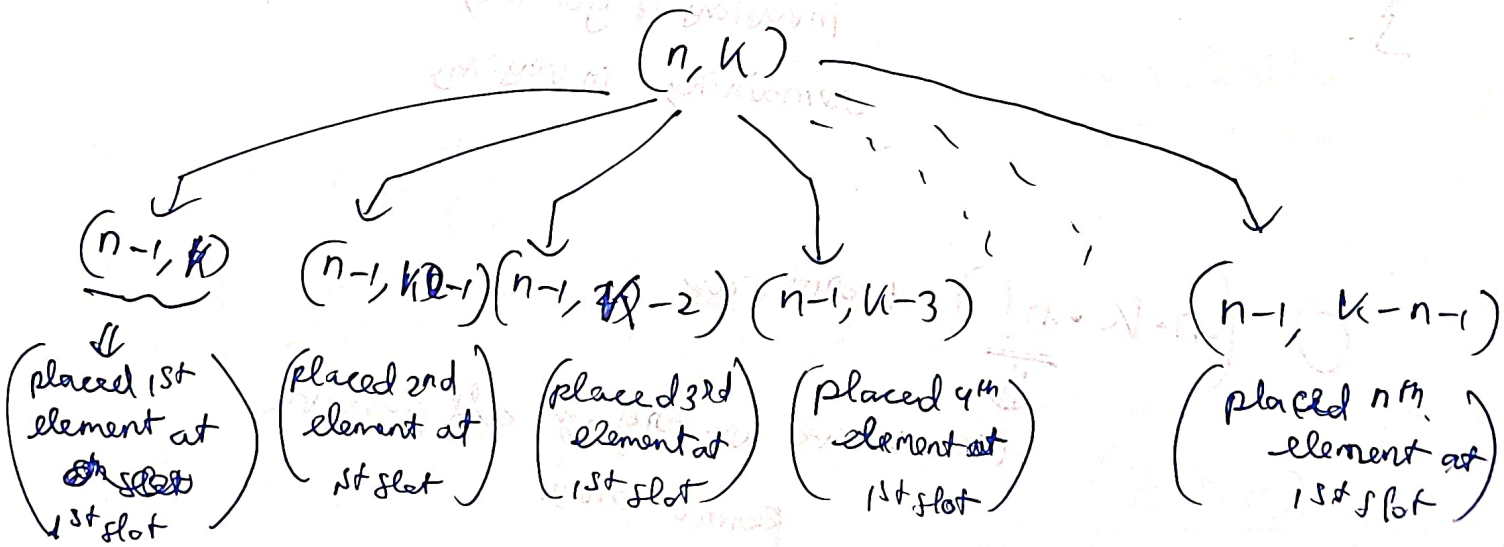
Q/ Does it matter any way which elements are remaining

You can treat it as → [1 2 3 4 5 ---- n-1] as

[2 3 4 5 6 ---- n]

it does not matter. what matters is the no of elements.

So for a given (n, k)



Depending on which element we are picking for the first slot decides the next state.

Now we got the recursive ~~step~~ state,

```
int f (int n, int k) {
  if (N <= 1) { return 1; }
  if (k == 0) { return 1; } → { No inversion, is sorted entry itself }
  if (k < 0) { return 0; } → { Neg inversion, is not possible }.
  ans = 0;
  for (i = 1; i <= N; i++) {
    ans = ans + f(n-1, k-i-1);
  }
}.
```

⇓

Based on what element we placed at the first slot, and the number of inversions it got me, I will place the remaining inversions

O $(n \cdot k \cdot n!)$ worst case

⇓

Because we are trying all possible permutations.

Lets introduce caching

int[][] cache = new int[N][K];

int f(n, k) {

~~if (n~~

we check if (cache[n][k] != -1) {

return cache[n][k];

}.

and store results in the cache at the end.

cache[n][k] = ans;

}

↳ Sowe visit (n·K) ~~steps~~ states each time,

And in each state we are doing a loop from (1 to N)

(n·K)·n → O(n²K) → time complexity.

O(n×K) Space complexity

(n²K)

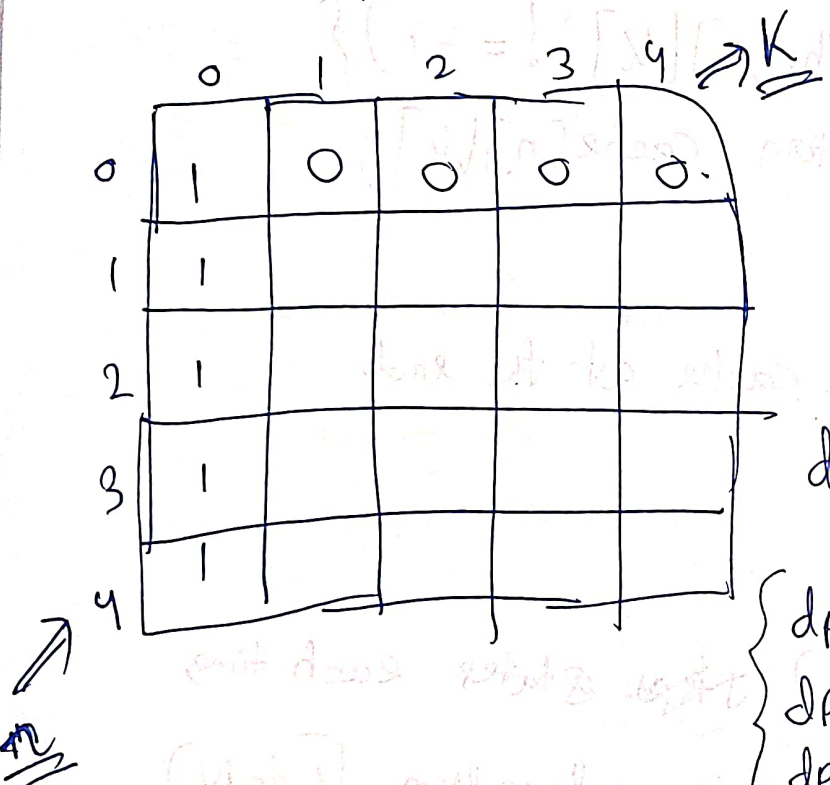
[1 ≤ (n, K) ≤ 1000] TLE

(P.T.O)

Neetcode then said, **Iterative dp with O(n×K) space**

~~try to solve~~

whenever you are not able to figure out how to optimize more on the recursive dp, try bottom-up iterative dp.



dp[n][K] represents how many arrays can be formed of [1-n] which has K inversions.

dp[0][0] = 0 elements in subarray with 0 inversions.

dp[1][0], dp[2][0], dp[3][0] → 0 inversions.

dp[0][1], dp[0][2], dp[0][3] → with 0 elements we cannot expect non-zero inversions

so 0

Now for any dp[n][K] ⇒

place 1st element at first spot and calculate for n-1 elements = dp[n-1][K] (no inversion detected.)

placing 2nd element at first spot and calculate n-1 elements $= dp[n-1][k-1]$

⇓ cause of 1 inversion.

placing 3rd element at first spot and calculate n-1 elements $= dp[n-1][k-2]$

⇓ cause 2 inversions introduced

⋮

Keep placing $dp[n-1][k-i]$ → $i \le k$ keep placing inversions like this.

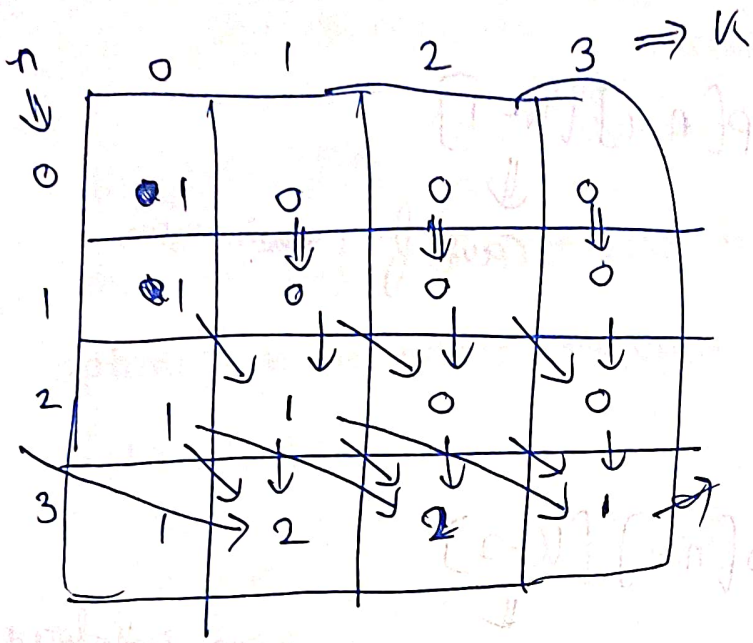
Code.

⇓ After Base case initialisation.

```
for (int i = 1 to N) {
    for (j = 1 to K) {

        for (inv = 0; inv ≤ i; inv++) {
            dp[i][j] += dp[i-1][k-inv]
        }
    }
}
```

add an if condition for negative indexing



dp[3][3] = 1

3 2 1

[(3,1), (2,1), (3,2)]

(Next optimisation)

(Space optimisation)

if we see properly we are only calculating ans for the i^{th} row, using the $(i-1)^{th}$ row.

So instead of maintaining a 2D array, why not maintain 2 arrays, one prev and one current

prev = [1, 0, 0, 0] ⇒ Base case condition one

cur = [0, 0, 0, 0].

prev [1, 1, 0, 0]

cur [1, 2, 2, 1]

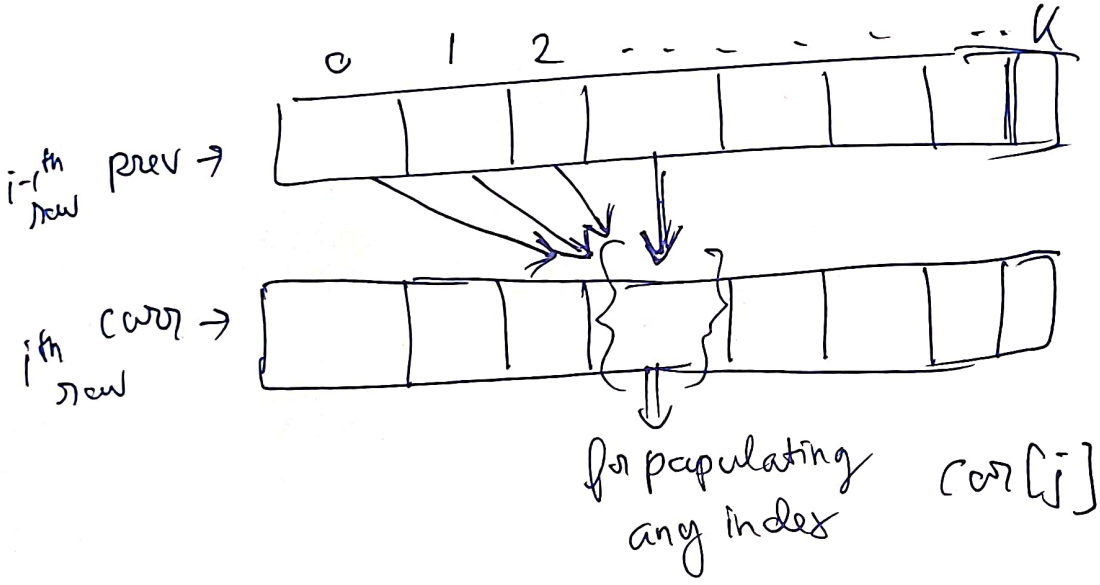
output cur[K].

```
for (i = 1 to N){
    for (j = 0 to K){
        for (inv = 0 to N-1){
            if (j - inv >= 0)
                cur[j] += prev[j - inv]
        }
    }
    prev = cur;
}
```

In this optimisation, we have reduced space from $O(N*K) \rightarrow O(K)$ ~~So from this we see that~~.

(Last optimisation.) [PrefixSum]

for every row i, if you notice



$$curr[j] = prev[j] + prev[j-1] + \ldots\ldots + prev[j-(N-1)]$$

which is nothing but prefix sum.

$(prev[j-(N-1)], \ldots\ldots, prev[j-1], prev[j])$.

⇓ if asked

So if in prev[j] we keep storing prefix sum.

$$curr[j] = prev[j] - prev[j-i]$$

⇓ for the ith row.