Dear reader, welcome to the next problem in the Recursion & Backtracking section named ‘[***Permutations - 1***](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/permutation-i-official/ojquestion)’.

You are standing at a very crucial point of your journey to master data structures and algorithms. Next series of problems are the conceptual building blocks of recursion and backtracking.

Trust me, if you will conquer these sets of problems, then you will not face any problem in solving problems based on recursion and backtracking.

***Problem Statement:***

* You are given a number of boxes (nboxes) and a number of non-identical items (ritems).
* You are required to place the items in those boxes and print all such configurations possible.
* Items are numbered from 1 to ritems.
* Note 1: Number of boxes is greater than number of items, hence some of the boxes may remain empty.
* Note 2: Check out the output format and write the recursive code as it is intended without changing signature. The judge can't force you but intends you to teach a concept.

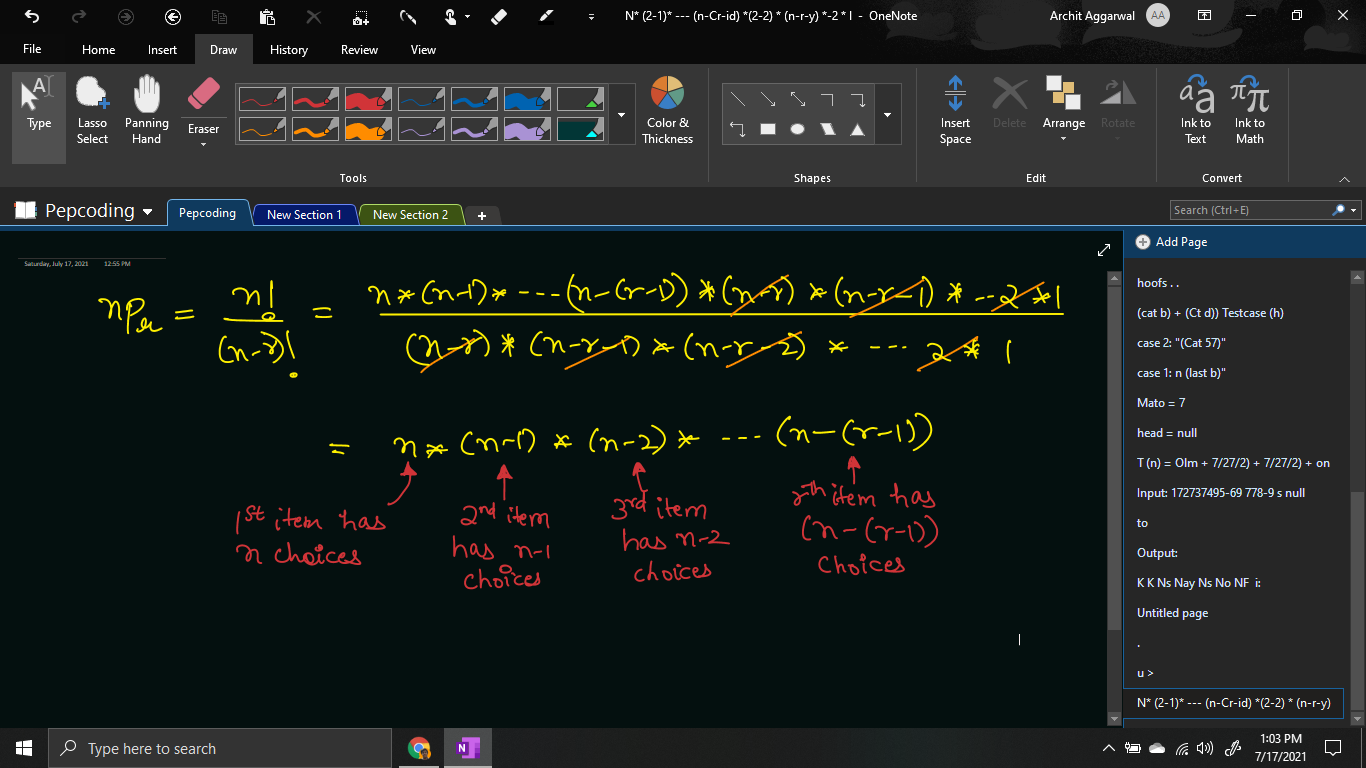
***Example:***

*Input*: Number of boxes (nboxes) = 4, number of non-identical items (ritems) = 3

*Output*: [1230, 1203, 1320, 1023, 1302, 1032, 2130, 2103, 3120, 0123, 3102, 0132, 2310, 2013, 3210, 0213, 3012, 0312, 2301, 2031, 3201, 0231, 3021, 0321]

***Solution***

Do you remember the ***permutation coefficient*** formula which you probably learned in high school mathematics?



**How** can we use the above formula to place ‘r’ non-identical items in ‘n’ boxes such that n >= r?

As you can see from the above formula, the 1st item has n choices, whether to be placed in any box. Now, after the 1st item is placed, since there are n-1 boxes empty, hence the 2nd item has n-1 choices of which box to be placed. Similarly, the 3rd item has n-3 choices and so on.

We will simulate this process until we have used all the r items. Initially, all boxes will be empty (denoted by box[i] = 0 for all i = 0 to n-1).

Let us define the faith, expectation of the recursive function and then derive a recursive relation by meeting expectation with faith.

**Expectation**: We expect that our function will give us all the permutations of r non-identical items by placing them in n boxes. We will expect from the function that it will place the current item ci, in any one empty box.

permutations(int[] boxes, int ci, int ti)

**Faith**: After placing the current item ci in any one box, we will keep faith on our recursive function, that it knows how to generate the permutation of the rest of the items in the empty boxes. Hence, we will keep faith that we can place all the rest items from (ci + 1 to ti) in some boxes.

permutations(int[] boxes, ci + 1, ti)

**Meeting Expectation with Faith:**

As mentioned above, for the current item ci, we will try to place it in one of the empty boxes. Hence, we will run a loop over all the boxes, checking whether the box is empty or not.

If the box is empty, then we will place the current item in the box, in the ***edge-pre area*** by doing box[i] = ci, and call for the recursive function for the next item (ci + 1).

But, please keep in mind that after returning from the recursive function in the **edge-post area**, we will update the box back to empty, i.e. box[i] = 0.

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

if(boxes[i] == 0){

boxes[i] = ci;

permutations(boxes, ci + 1, ti);

boxes[i] = 0;

}

}

Now, the only thing remaining is … the **BASE CASE**.

We should stop if we have placed all the total number of items (ti) in some boxes. Hence, when the current item (ci) becomes greater than ti, then we will print the permutation and return.

if(ci > ti){

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

System.out.print(boxes[i]);

}

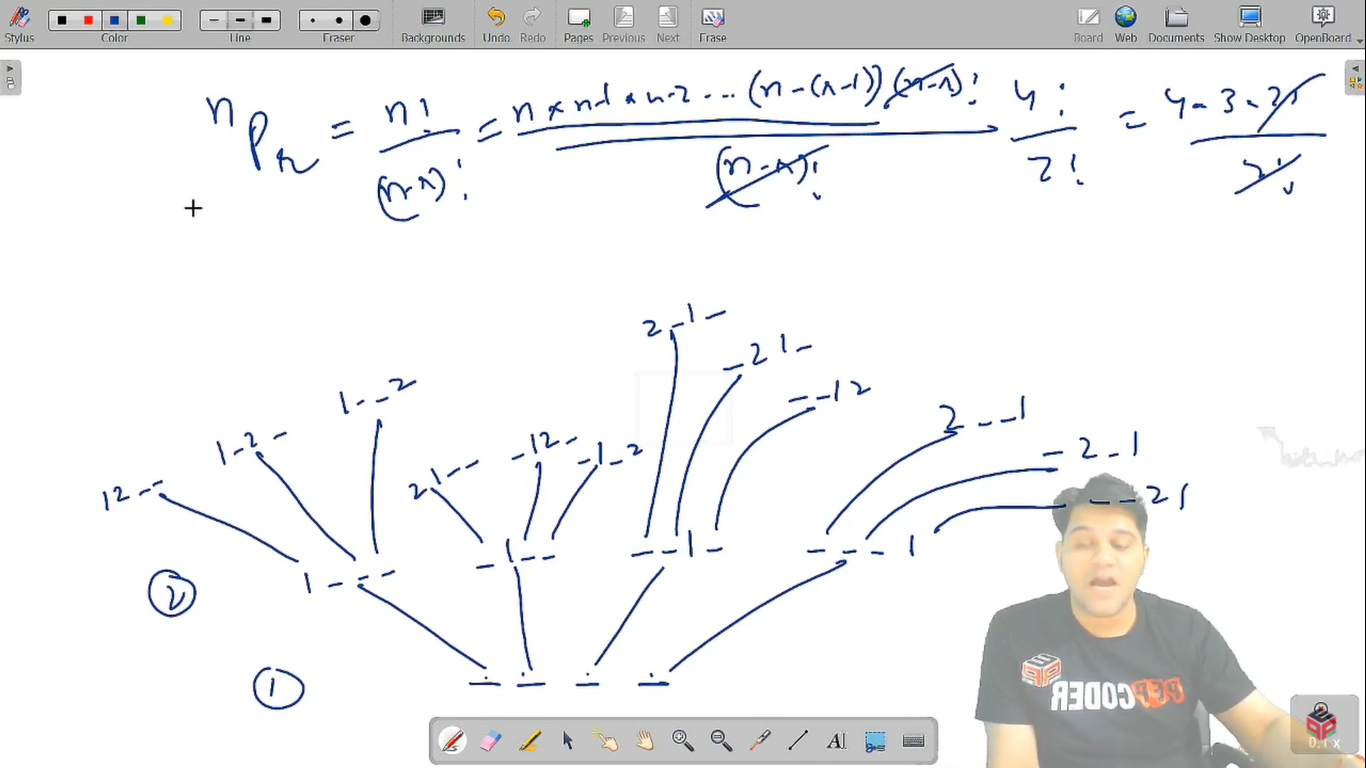
System.out.println();

return;

}

In recursion tree:

* The ***LEVELS are equivalent to ITEMS***.
* The ***CHOICES/EDGES are equivalent to BOXES***.

****

**Implementation (Java Code)**

import java.io.\*;

import java.util.\*;

public class Main {

public static void permutations(int[] boxes, int ci, int ti){

if(ci > ti){

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

System.out.print(boxes[i]);

}

System.out.println();

return;

}

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

if(boxes[i] == 0){

boxes[i] = ci;

permutations(boxes, ci + 1, ti);

boxes[i] = 0;

}

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new

BufferedReader(new InputStreamReader(System.in));

int nboxes = Integer.parseInt(br.readLine());

int ritems = Integer.parseInt(br.readLine());

permutations(new int[nboxes], 1, ritems);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=QKkHCS5bq0I&list=TLGGEstfbiQGn0ExNzA3MjAyMQ). Please refer to it for a better understanding of the algorithm and the implementation. Also, try to **dry run** the recursion by taking some examples and drawing the recursion tree.

* What is the ***time complexity*** of the above code?

Since, there are r levels/depth of the recursion tree, and at each level, we are exploring all the n boxes and finding the empty ones, hence the time complexity will be O(n \* n \* …. r times) = O(**nr**).

* What is the ***space complexity*** of the above code?

We are using the box array to store the permutations. If we consider the space taken by this array, then space complexity will be **O(n)**.

Otherwise, since we are using recursion which takes function call space, and there are at max r depth, recursion will take **O(r)** space.

Hope that you liked the article on ***’Permutations - 1’***.

Subscribe to Pepcoding’s youtube channel for more such amazing video content on Data Structures & Algorithms. You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by:

[Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)