Dear reader, welcome to the next problem in the Recursion & Backtracking section named ‘[***Combinations - 2***](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/combinations-2-official/ojquestion)’.

Please, I request you to solve the previous problems ‘[Permutations - 1](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/permutation-i-official/ojquestion)’, ‘[Combinations - 1](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/combinations-i-official/ojquestion)’ and ‘[Permutations - 2](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/recursion-and-backtracking/permutations-ii-official/ojquestion)’ before jumping on to this problem. Infact, I recommend you to watch these sets of problems in given order and try to complete the entire set together.

***Problem Statement:***

* You are given a number of boxes (nboxes) and a number of identical items (ritems).
* You are required to place the items in those boxes and print all such configurations possible.
* Items are identical and all of them are named 'i'.
* 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:***

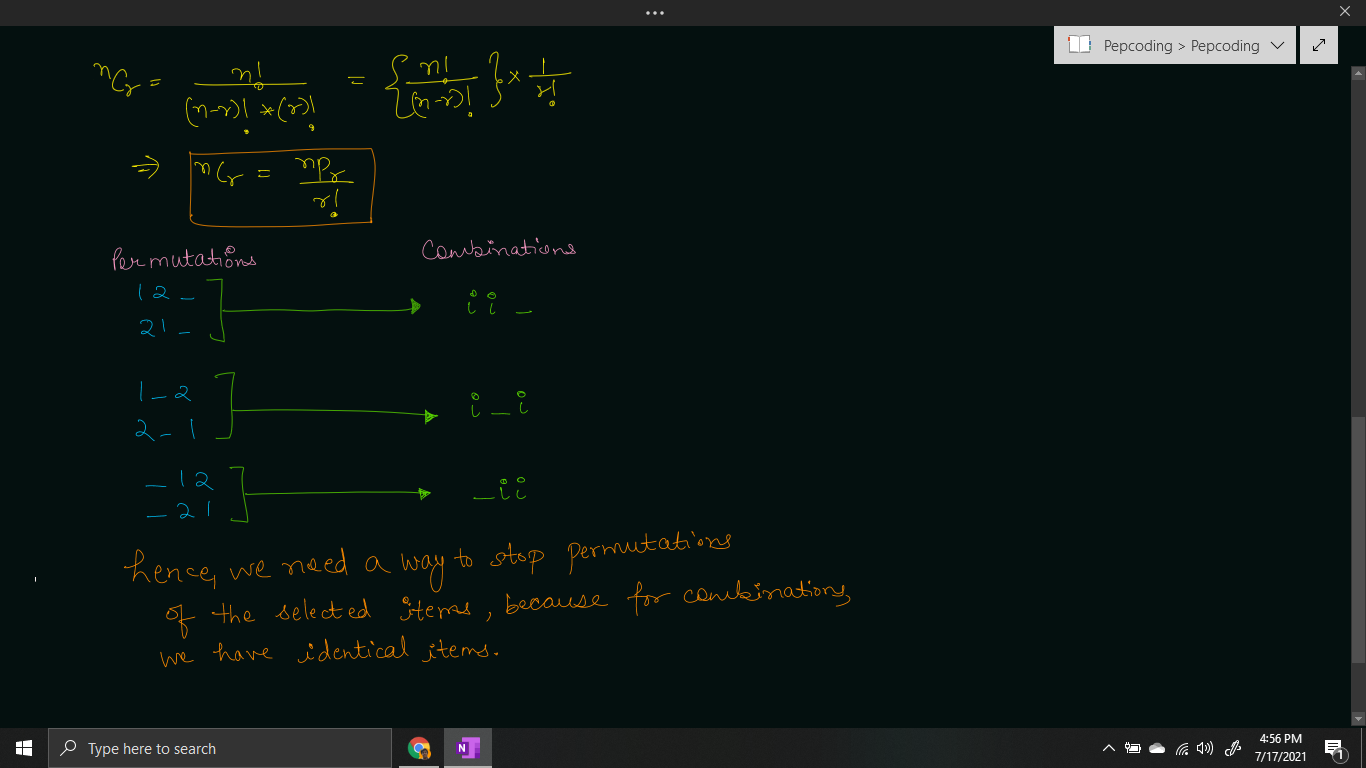
*Input*: Number of boxes (nboxes) = 5, number of identical items (ritems) = 3

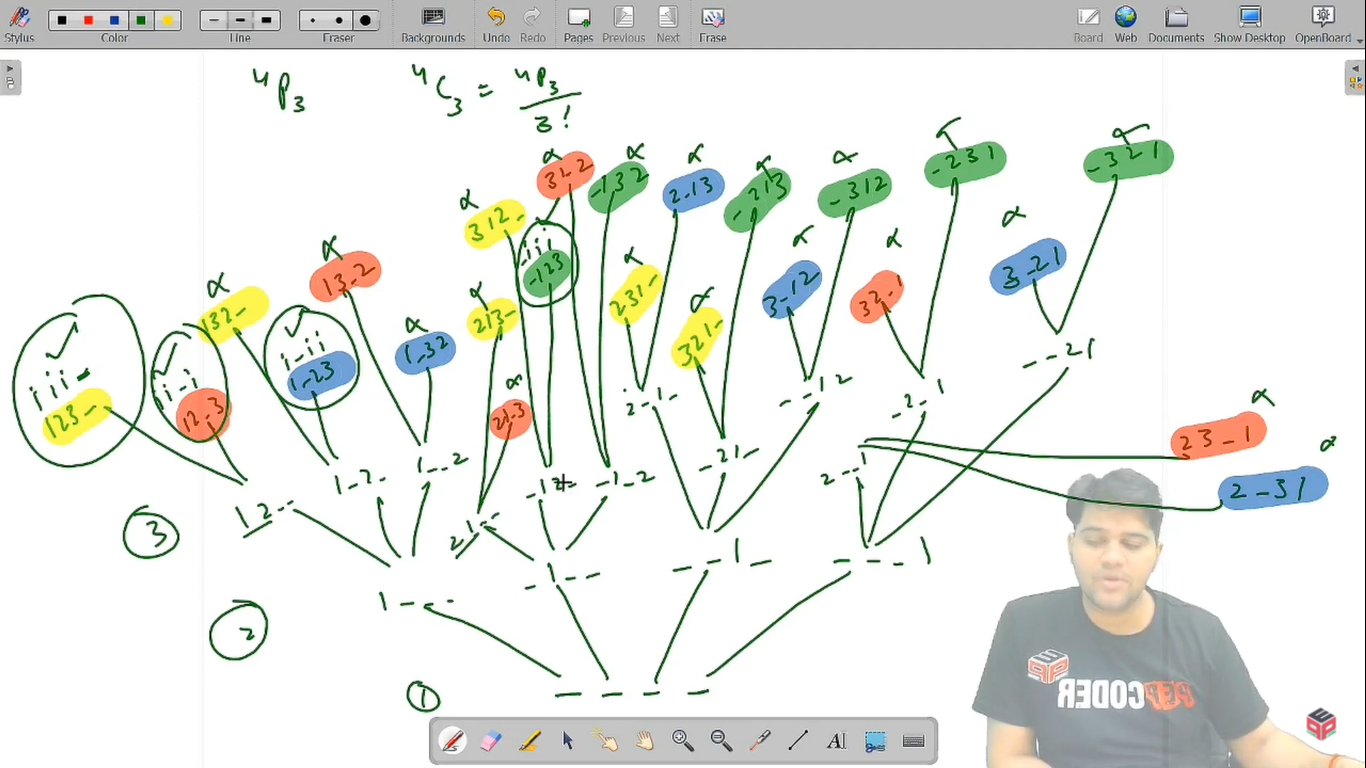
*Output*: [ iii-- , ii-i- , ii--i , i-ii- , i-i-i , i--ii , -iii- , -ii-i , -i-ii , --iii ]

***Solution***

In the previous solution for the same combination problem, we used LEVELS as boxes and choices as placing an item or not.

But in this article, we will try to find a solution by deriving a relation between permutation coefficient and ***binomial coefficient***.





If we are having 123 as a permutation, then we should eliminate all possibilities of 132 , 213, 231, 312 and 321, because in this case all items 1,2 and 3 are identical (= i).

**Q)** So, how can we stop the remaining permutations?

**R)** If you analyze carefully, if we generate all the permutations with **items in increasing order of their values** only, and discard all the other permutations, then we will be able to find combinations using permutations.

Hence, if the current item cb is placed in say box number b, then the next item cb + 1 should be placed in boxes in right of b, i.e. in any box from [b + 1, n] only.

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 combinations of r 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. Here, we also need the index of the last box lb.

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

**Faith**: After placing the current item ci in box number b, we will keep faith on our recursive function, that it knows how to generate the combinations of the rest of the items in the empty boxes in range [b+1, n]. Hence, we will keep faith that we can place all the rest items from (ci + 1 to ti) in boxes (b+1 to n).

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

**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 the boxes after the last box, 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[b] = 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[b] = 0.

for(int b = lb + 1; b < boxes.length; b++){

if(boxes[b] == 0){

boxes[b] = ci;

combinations(boxes, ci + 1, ti, b);

boxes[b] = 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 combination and return.

But, please note that values in boxes array are integer values (1, 2, 3, … r), but we need ’i’ for filled boxes (identical items). Hence, if boxes[i] = 0, then we will print “-” (empty) else i (filled).

if(ci > ti){

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

System.out.print(boxes[i] == 0? "-" : "i");

}

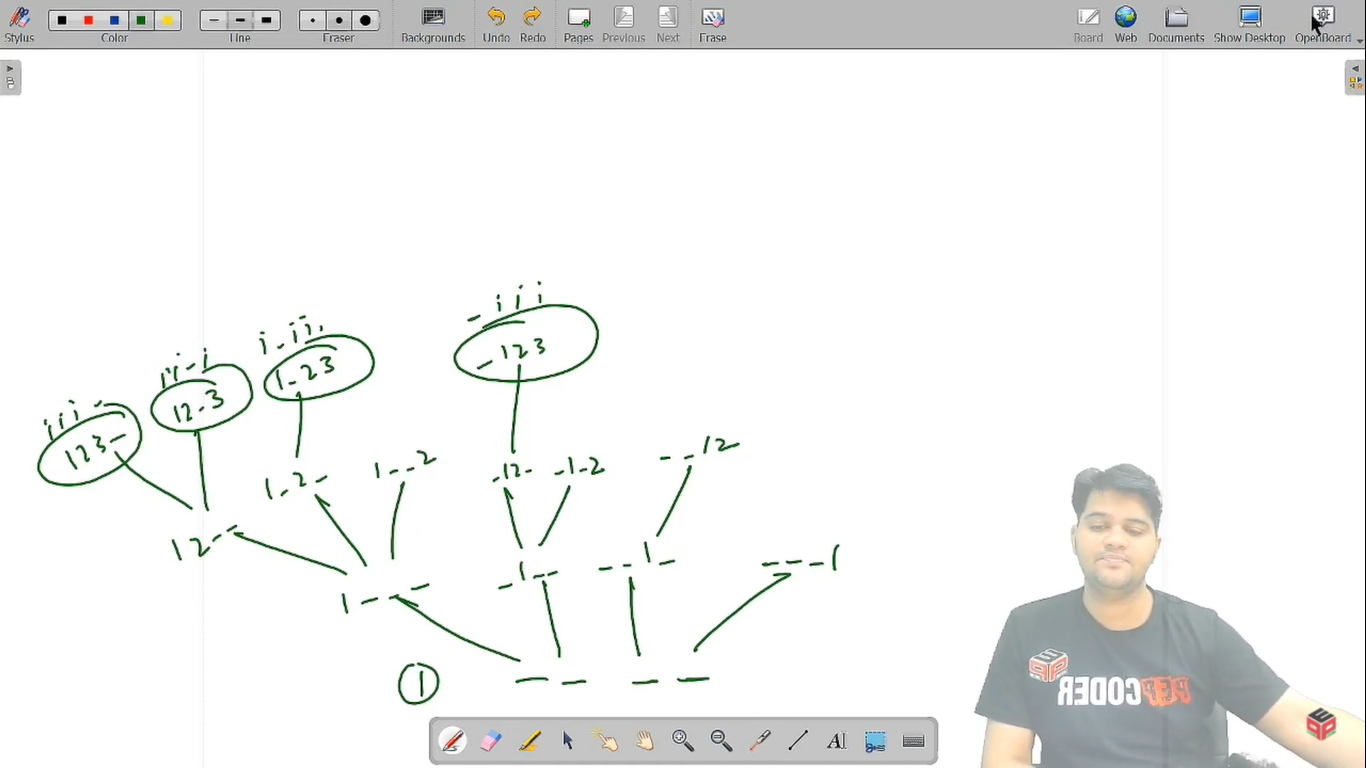
System.out.println();

return;

}

In recursion tree:

* The ***LEVELS are equivalent to ITEMS*** *(but in increasing order only)*.
* The ***CHOICES/EDGES are equivalent to BOXES***.

****

**Java Code**

import java.io.\*;

import java.util.\*;

public class Main {

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

if(ci > ti){

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

System.out.print(boxes[i] == 0? "-" : "i");

}

System.out.println();

return;

}

for(int b = lb + 1; b < boxes.length; b++){

if(boxes[b] == 0){

boxes[b] = ci;

combinations(boxes, ci + 1, ti, b);

boxes[b] = 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());

combinations(new int[nboxes], 1, ritems, -1);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=f6cL-VMIfTY&list=TLGGJFKMpKAmgoUxNzA3MjAyMQ). Please refer to it for a better understanding of the algorithm and the implementation.

* 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 at max 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 combinations. 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 **‘*Combinations - 2’***.

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/)