# Question

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

You may return the answer in **any order**.

**Example 1:**

**Input:** n = 4, k = 2

**Output:**

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

**Example 2:**

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

**Output:** [[1]]

**Constraints:**

* 1 <= n <= 20
* 1 <= k <= n

# Solution

#### **Approach 1: Backtracking**

**Algorithm**

[Backtracking](https://en.wikipedia.org/wiki/Backtracking) is an algorithm for finding all solutions by exploring all potential candidates. If the solution candidate turns to be not a solution (or at least not the last one), backtracking algorithm discards it by making some changes on the previous step, i.e. backtracks and then try again.

Here is a backtrack function which takes a first integer to add and a current combination as arguments backtrack(first, curr).

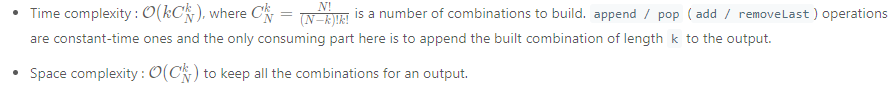
* If the current combination is done - add it to output.
* Iterate over the integers from first to n.
  + Add integer i into the current combination curr.
  + Proceed to add more integers into the combination : backtrack(i + 1, curr).
  + Backtrack by removing i from curr.

**Implementation**

1 / 12

|  |
| --- |
| **class Solution {**  **List<List<Integer>> output = new LinkedList();**  **int n;**  **int k;**  **public void backtrack(int first, LinkedList<Integer> curr) {**  **// if the combination is done**  **if (curr.size() == k)**  **output.add(new LinkedList(curr));**  **for (int i = first; i < n + 1; ++i) {**  **// add i into the current combination**  **curr.add(i);**  **// use next integers to complete the combination**  **backtrack(i + 1, curr);**  **// backtrack**  **curr.removeLast();**  **}**  **}**  **public List<List<Integer>> combine(int n, int k) {**  **this.n = n;**  **this.k = k;**  **backtrack(1, new LinkedList<Integer>());**  **return output;**  **}**  **}** |

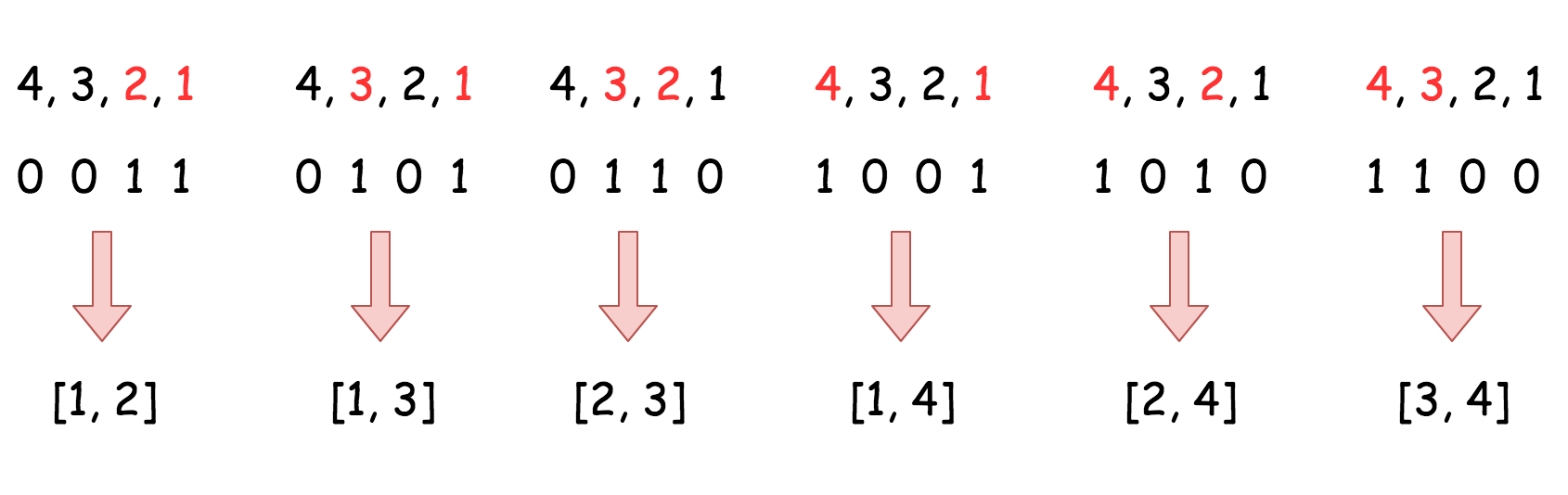
**Complexity Analysis**



#### **Approach 2: Lexicographic (binary sorted) combinations**

**Intuition**

The idea here is not just to get the combinations but to generate them in a lexicographic sorted order.



**Algorithm**

The algorithm is quite straightforward :

* Initiate nums as a list of integers from 1 to k. Add n + 1 as a last element, it will serve as a sentinel. Set the pointer in the beginning of the list j = 0.
* While j < k :
  + Add the first k elements from nums into the output, i.e. all elements but the sentinel.
  + Find the first number in nums such that nums[j] + 1 != nums[j + 1] and increase it by one nums[j]++ to move to the next combination.

**Implementation**

|  |
| --- |
| class Solution {  public List<List<Integer>> combine(int n, int k) {  // init first combination  LinkedList<Integer> nums = new LinkedList<Integer>();  for(int i = 1; i < k + 1; ++i)  nums.add(i);  nums.add(n + 1);  List<List<Integer>> output = new ArrayList<List<Integer>>();  int j = 0;  while (j < k) {  // add current combination  output.add(new LinkedList(nums.subList(0, k)));  // increase first nums[j] by one  // if nums[j] + 1 != nums[j + 1]  j = 0;  while ((j < k) && (nums.get(j + 1) == nums.get(j) + 1))  nums.set(j, j++ + 1);  nums.set(j, nums.get(j) + 1);  }  return output;  }  } |

**Complexity Analysis**

