77. Combinations

March 22, 2019 | 24.4K views

Average Rating: 4.38 (21 votes)

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

Example:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Solution

Approach 1: Backtracking

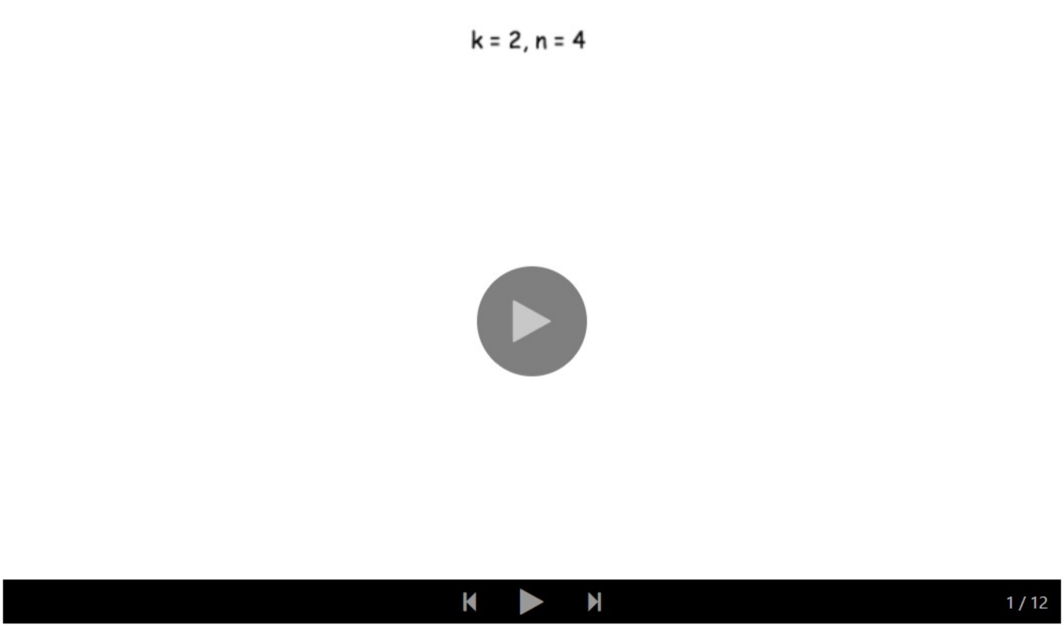
Algorithm

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



Java Python Copy

```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        def backtrack(first = 1, curr = []):
            # if the combination is done
            if len(curr) == k:
                output.append(curr[:])
            for i in range(first, n + 1):
                # add i into the current combination
                curr.append(i)
                # use next integers to complete the combination
                backtrack(i + 1, curr)
                # backtrack
                curr.pop()

        output = []
        backtrack()
        return output
```

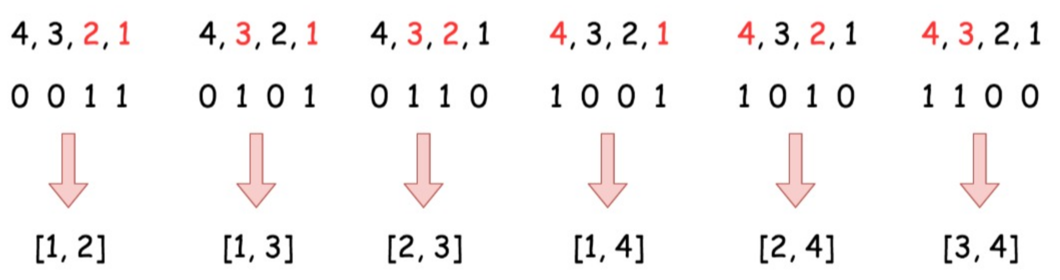
Complexity Analysis

- Time complexity : $\mathcal{O}(k C_N^k)$, where $C_N^k = \frac{N!}{(N - k)! k!}$ is a number of combinations to build. `append / pop` (`add / removeLast`) operations are constant-time ones and the only consuming part here is to append the built combination of length `k` to the output.
- Space complexity : $\mathcal{O}(C_N^k)$ to keep all the combinations for an output.

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

Java Python Copy

```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        # init first combination
        nums = list(range(1, k + 1)) + [n + 1]

        output, j = [], 0
        while j < k:
            # add current combination
            output.append(nums[:k])
            # increase first nums[j] by one
            # if nums[j] + 1 != nums[j + 1]
            j = 0
            while j < k and nums[j + 1] == nums[j] + 1:
                nums[j] = j + 1
                j += 1
            nums[j] += 1

        return output
```

Complexity Analysis

- Time complexity : $\mathcal{O}(k C_N^k)$, where $C_N^k = \frac{N!}{(N - k)! k!}$ is a number of combinations to build.

  The external while loop is executed C_N^k times since the number of combinations is C_N^k. The inner while loop is performed C_{N-j}^{k-j} times for a given `j`. In average over C_N^k visits from the external loop that results in less than one execution per visit.
  Hence the most consuming part here is to append each combination of length k (C_N^k combinations in total) to the output, that takes $\mathcal{O}(k C_N^k)$ time.

  You could notice that the second algorithm is much faster than the first one despite they both have the same time complexity. It's a consequence of dealing with the recursive call stack frame for the first algorithm, and the effect is much more pronounced in Python than in Java.

- Space complexity : $\mathcal{O}(C_N^k)$ to keep all the combinations for an output.

Links

Donald E. Knuth, The Art of Computer Programming, 4A (2011)