

LeetCode

ExploreProblemsMockContestArticlesDiscussStore

Articles > 77. Combinations

PreviousNext

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  $1 \dots n$ .

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

k = 2, n = 4

JavaPython

Copy

```
1 class Solution:
2     def combine(self, n: int, k: int) -> List[List[int]]:
3         def backtrack(first = 1, curr = []):
4             # if the combination is done
5             if len(curr) == k:
6                 output.append(curr[:])
7             for i in range(first, n + 1):
8                 # add i into the current combination
9                 curr.append(i)
10                # use next integers to complete the combination
11                backtrack(i + 1, curr)
12                # backtrack
13                curr.pop()
14
15        output = []
16        backtrack()
17        return output
```

Complexity Analysis

- Time complexity :  $\mathcal{O}(kC_N^k)$ , 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.

4, 3, 2, 14, 3, 2, 14, 3, 2, 14, 3, 2, 14, 3, 2, 14, 3, 2, 1

0 0 1 10 1 0 10 1 1 01 0 0 11 0 1 01 1 0 0

↓↓↓↓↓

[1, 2][1, 3][2, 3][1, 4][2, 4][3, 4]

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

JavaPython

Copy

```
1 class Solution:
2     def combine(self, n: int, k: int) -> List[List[int]]:
3         # init first combination
4         nums = list(range(1, k + 1)) + [n + 1]
5
6         output, j = [], 0
7         while j < k:
8             # add current combination
9             output.append(nums[:k])
10            # increase first nums[j] by one
11            # if nums[j] + 1 != nums[j + 1]
12            j = 0
13            while j < k and nums[j + 1] == nums[j] + 1:
14                nums[j] = j + 1
15                j += 1
16            nums[j] += 1
17
18        return output
```

Complexity Analysis

- Time complexity :  $\mathcal{O}(kC_N^k)$ , 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  $\mathcal{O}(kC_N^k)$  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)

Rate this article: ★★★★★

PreviousNext


Comments: 18Sort By ▾



Type comment here... (Markdown is supported)

Preview

Post



dylan20★196

March 29, 2019 9:00 PM


solution 1 should return after appending a complete combination. otherwise the combination will grow beyond size K

17

Share

Reply

SHOW 5 REPLIES



mircules★43


July 2, 2019 5:50 PM

Shouldn't the backtracking approach's space complexity be the same as the time complexity of  $O(k * nCk)$ ? We have  $nCk$  subsets and they're all  $k$  in size

6

Share

Reply



purplebeth★6

June 10, 2020 11:16 PM


why do we need to create a shallow copy of curr in the python solution? (curr[:])

2

Share

Reply

SHOW 1 REPLY



tom53★26

July 27, 2019 11:20 AM


Add return condition for approach1 (prune unused call stack).

2

Share

Reply

SHOW 1 REPLY



shakeri★29


July 18, 2019 11:25 PM

The space complexity of backtracking should be  $O(k+Ckn)$

1

Share

Reply



wonder2008★1


April 13, 2019 11:28 PM

In both approaches, for the space complexity, should it also be scaled by a factor of  $k$ ?

1

Share

Reply



glfox★1


May 27, 2019 5:02 AM

shouldn't the time complex of combination is : 2 to the power of K, the above is more like permutation time complexity

0

Share

Reply



zhang-peter★26


March 26, 2019 4:10 PM

why the graph in Approach 2 is 4,3,2,1 not 1,2,3,4?  
4,3,2,1 takes me a long time to understand it.

0

Share

Reply



user4550C★59


July 11, 2019 1:18 AM

what time complexity for approach 1 is not the summation from  $(n,1)$  to  $(n, k)$

0

Share

Reply



ShannonL★0

July 1, 2020 5:47 AM

In Approach 1, why output.append(curr[:]), not output.append(curr)? I know lists are referred by references, but cannot distinguish the use case in problems like this.

0

Share

Reply

<12>