216. Combination Sum III

Solution

Approach 1: Backtracking

Intuition

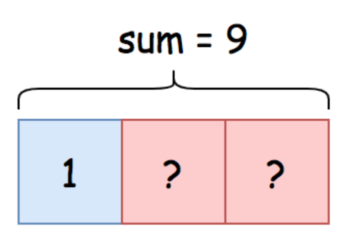
The problem asks us to come up with some fixed-length combinations that meet certain conditions.

To solve the problem, it would be beneficial to build a combination by hand.

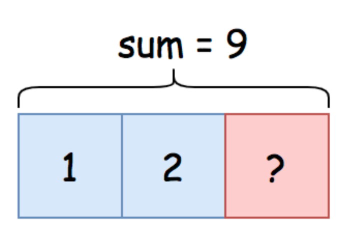
If we represent the combination as an array, we then could fill the array ***one element at a time***.

For example, given the input $k = 3$ and $n = 9$, *i.e.* the size of the combination is 3, and the sum of the digits in the combination should be 9. Here are a few steps that we could do:

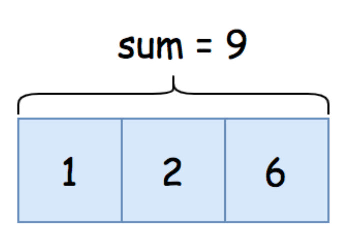
- 1). We could pick a digit for the ***first*** element in the combination. Initially, the list of candidates is `[1, 2, 3, 4, 5, 6, 7, 8. 9]` for any element in the combination, as stated in the problem. Let us pick `1` as the first element. The current combination is `[1]`.



- 2). Now that we picked the first element, we have two more elements to fill in the final combination. Before we proceed, let us review the conditions that we should fullfil for the next steps.
  - Since we've already picked the digit `1`, we should exclude the digit from the original candidate list for the remaining elements, in order to ensure that the combination does not contain any ***duplicate*** digits, as required in the problem.
  - In addition, the sum of the remaining two elements should be $9 - 1 = 8$.
- 3). Based on the above conditions, for the second element, we could have several choices. Let us pick the digit `2`, which is not a duplicate of the first element, plus it does not exceed the desired sum (*i.e.* 8) neither. The combination now becomes `[1, 2]`.

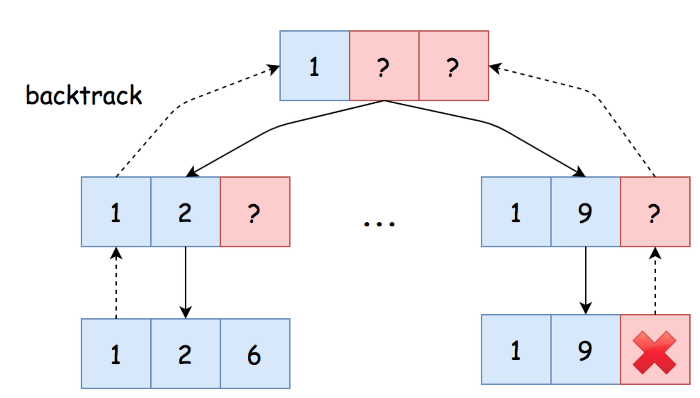


- 4). Now for the third element, with all the constraints, it leaves us no choice but to pick the digit `6` as the final element in the combination of `[1, 2, 6]`.



- 5). As we mentioned before, for the second element, we could have several choices. For instance, we could have picked the digit `3`, instead of the digit `2`. Eventually, it could *lead* us to another solution as `[1, 3, 5]`.
- 6). As one can see, for each element in the combination, we could ***revisit*** our choices, and ***try out*** other possibilities to see if it leads to a valid solution.

If you have followed the above steps, it should become *evident* that ***backtracking*** would be the technique that we could use to come up an algorithm for this problem.



> Indeed, we could resort to *backtracking*, where we try to fill the combination **one element at a step**. Each choice we make at certain step might lead us to a final solution. If not, we simply revisit the choice and try out other choices, *i.e.* backtrack.

Algorithm

There are many ways to implement a backtracking algorithm. One could also refer to our Explore card (https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/) where we give some examples of backtracking algorithms.

To implement the algorithm, one could literally follow the steps in the Intuition section. However, we would like to highlight a key ***trick*** that we employed, in order to ensure the ***non-redundancy*** among the digits within a single combination, as well as the ***non-redundancy*** among the combinations.

> The trick is that we pick the candidates ***in order***. We treat the candidate digits as a list with order, *i.e.* `[1, 2, 3, 4, 5, 6, 7, 8. 9]`. At any given step, once we pick a digit, *e.g.* `6`, we will not consider any digits before the chosen digit for the following steps, *e.g.* the candidates are reduced down to `[7, 8, 9]`.

With the above strategy, we could ensure that a digit will never be picked twice for the same combination. Also, all the combinations that we come up with would be unique.

Here are some sample implementations based on the above ideas.

Java | Python3 | Copy

```java
class Solution {
    protected void backtrack(int remain, int k,
            LinkedList<Integer> comb, int next_start,
            List<List<Integer>> results) {

        if (remain == 0 && comb.size() == k) {
            // Note: it's important to make a deep copy here,
            // Otherwise the combination would be reverted in other branch of backtracking.
            results.add(new ArrayList<Integer>(comb));
            return;
        } else if (remain < 0 || comb.size() == k) {
            // Exceed the scope, no need to explore further.
            return;
        }

        // Iterate through the reduced list of candidates.
        for (int i = next_start; i < 9; ++i) {
            comb.add(i + 1);
            this.backtrack(remain - i - 1, k, comb, i + 1, results);
            comb.removeLast();
        }
    }

    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        LinkedList<Integer> comb = new LinkedList<Integer>();
```

Complexity Analysis

Let K be the number of digits in a combination.

- Time Complexity: $\mathcal{O}(\frac{9! \cdot K}{(9-K)!})$
    - In a worst scenario, we have to explore all potential combinations to the very end, *i.e.* the sum n is a large number ($n > 9 * 9$). At the first step, we have 9 choices, while at the second step, we have 8 choices, so on and so forth.
    - The number of exploration we need to make in the worst case would be $P(9, K) = \frac{9!}{(9-K)!}$, assuming that $K <= 9$. By the way, K cannot be greater than 9, otherwise we cannot have a combination whose digits are all unique.
    - Each exploration takes a constant time to process, except the last step where it takes $\mathcal{O}(K)$ time to make a copy of combination.
    - To sum up, the overall time complexity of the algorithm would be $\frac{9!}{(9-K)!} \cdot \mathcal{O}(K) = \mathcal{O}(\frac{9! \cdot K}{(9-K)!})$.
- Space Complexity: $\mathcal{O}(K)$
    - During the backtracking, we used a list to keep the current combination, which holds up to K elements, *i.e.* $\mathcal{O}(K)$.
    - Since we employed recursion in the backtracking, we would need some additional space for the function call stack, which could pile up to K consecutive invocations, *i.e.* $\mathcal{O}(K)$.
    - Hence, to sum up, the overall space complexity would be $\mathcal{O}(K)$.
    - **Note that**, we did not take into account the space for the final results in the space complexity.

Comments: 12

Sort By

Type comment here... (Markdown is supported)

Preview | Post

cedrus (/cedrus) ★ 12 September 12, 2020 4:21 AM

(/cedrus)

I think this problem involves combination. In terms of combination, the time complexity is O(C(9,k)) = O(9!/(k! * (9-k)!). There is no duplicate in each combination, and the order of numbers doesn't matter.
It can be faster since all numbers sum up to n, and the case numbers added up more than n can return earlier.

12 Share Reply

SHOW 1 REPLY

clock330 (/clock330) ★ 49 September 6, 2020 10:12 AM

(/clock330)

I am confused by the time complexity analysis:

> At the first step, we have 9 choices, while at the second step, we have 8 choices, so on and so forth.

Read More

9 Share Reply

SHOW 4 REPLIES

Notes

Quick Navigation

magks (/magks) ★ 158 September 13, 2020 8:41 PM

In the python version, line 7 -- why is it necessary to `append(list(comb))` rather than just `append(comb)` ?

I tried it without wrapping comb and see that the result is appending an empty list to results but does anyone know why this is? Both `type(comb)` and `type(list(comb))` return <class 'list'> and also

Read More

1 Share Reply

SHOW 2 REPLIES

snibbets (/snibbets) ★ 29 September 12, 2020 3:26 PM

The number of steps should actually be bounded by a constant because if `n > 45` the result is always an empty list. If the algorithm returns `[]` whenever `n > 45`, the time complexity would be O(1).

1 Share Reply

KaitoKid56 (/kaitokid56) ★ 63 September 12, 2020 10:17 AM

Can anyone tell me why are we removing the last element on line 19?
Is it that in backtracking we have to remove ONLY the last element?

1 Share Reply

SHOW 5 REPLIES

alvin-777 (/alvin-777) ★ 213 September 12, 2020 9:44 AM

I don't know how should I estimate the time complexity of my solution below:

```
vector<vector<int>> combinationSum3(int k, int n) {
    if (!k || k > 9) return {};
    const vector<vector<int>> REF{{0, 1, 3, 6, 10, 15, 21, 28, 36, 45},
```

Read More

1 Share Reply

mdabarik (/mdabarik) ★ 59 September 12, 2020 8:13 AM

I solve another problem of my favorite topics Backtracking! It's helps me improve my critical problem solving skills.

1 Share Reply

133c7 (/133c7) ★ 82 September 15, 2020 5:11 AM

Wouldn't the time complexity be simply `~O(9Ck)` rather than `~O(9Ck * k)` ? My reasoning was that, although it takes `~O(k)` time to deeply copy the combination, not every branch in the recursion tree will need to copy the combination. In fact, it seemed to me that only a very small number of branch would end up copying the combination, as most branch will end up not arriving at the answer. But technically `~O(9Ck * k)` is a correct loose upperbound

Read More

0 Share Reply

cleydyr (/cleydyr) ★ 19 September 13, 2020 12:31 PM

Backtracking is not even necessary if you calculate the range of `i` correctly in each recursion step in order to trim out unnecessary (i.e., yielding an empty list) `(k, n, min)` combinations. Also, right from the beginning, you can return an empty list for some known values of `(k, n)` .

It's a matter of leveraging Gauss' idea of summing consecutive numbers

Read More

0 Share Reply

zhangbinqq53 (/zhangbinqq53) ★ 0 September 13, 2020 9:18 AM

'for i in range(next_start, 9)' in the solution, can '9' be replaced by 'min(9, remain)'?
C++ answer:

```
class Solution {
public:
```

Read More

0 Share Reply

‹ 1 2 ›

Notes