Previous Next

78. Subsets

Dec. 29, 2019 | 75.5K views

Average Rating: 4.46 (68 votes)

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

Solution

Solution Pattern

Let us first review the problems of Permutations / Combinations / Subsets, since they are quite similar to each other and there are some common strategies to solve them.

First, their solution space is often quite large:

- Permutations: $N!$.
- Combinations: $C_N^k = \frac{N!}{(N-k)!k!}$
- Subsets: 2^N, since each element could be absent or present.

Given their exponential solution space, it is tricky to ensure that the generated solutions are ***complete*** and ***non-redundant***. It is essential to have a clear and easy-to-reason strategy.

There are generally three strategies to do it:

- Recursion
- Backtracking
- Lexicographic generation based on the mapping between binary bitmasks and the corresponding permutations / combinations / subsets.

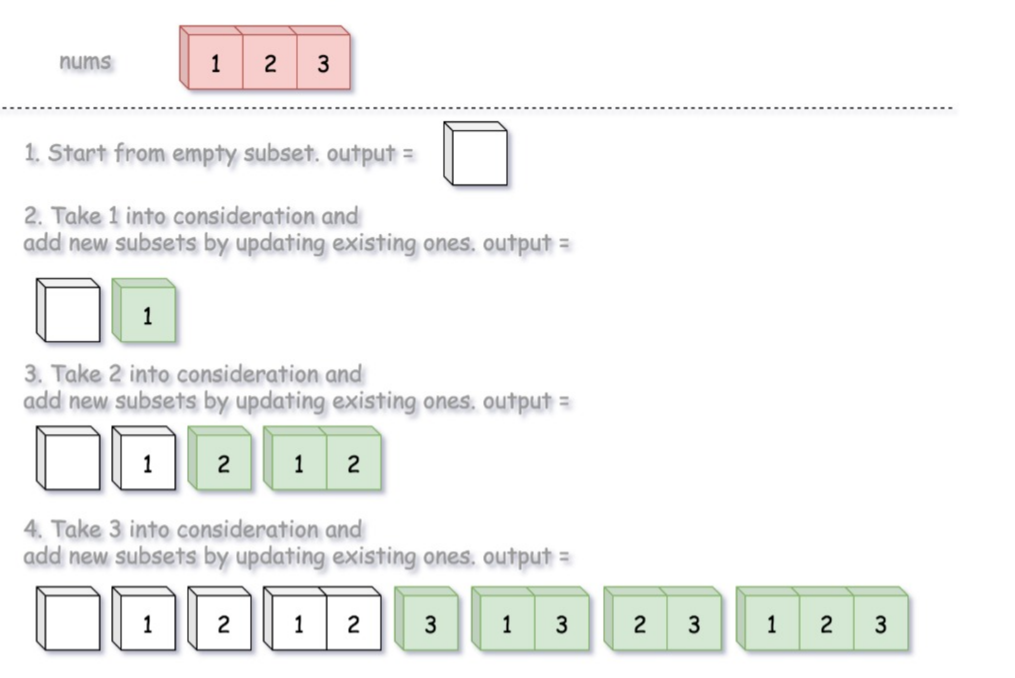
As one would see later, the third method could be a good candidate for the interview because it simplifies the problem to the generation of binary numbers, therefore it is easy to implement and verify that no solution is missing.

Besides, this method has the best time complexity, and as a bonus, it generates lexicographically sorted output for the sorted inputs.

Approach 1: Cascading

Intuition

Let's start from empty subset in output list. At each step one takes new integer into consideration and generates new subsets from the existing ones.



Implementation

Java Python Copy

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        output = [[]]

        for num in nums:
            output += [curr + [num] for curr in output]

        return output
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N \times 2^N)$ to generate all subsets and then copy them into output list.
- Space complexity: $\mathcal{O}(N \times 2^N)$. This is exactly the number of solutions for subsets multiplied by the number N of elements to keep for each subset.
  - For a given number, it could be present or absent (*i.e.* binary choice) in a subset solution. As as result, for N numbers, we would have in total 2^N choices (solutions).

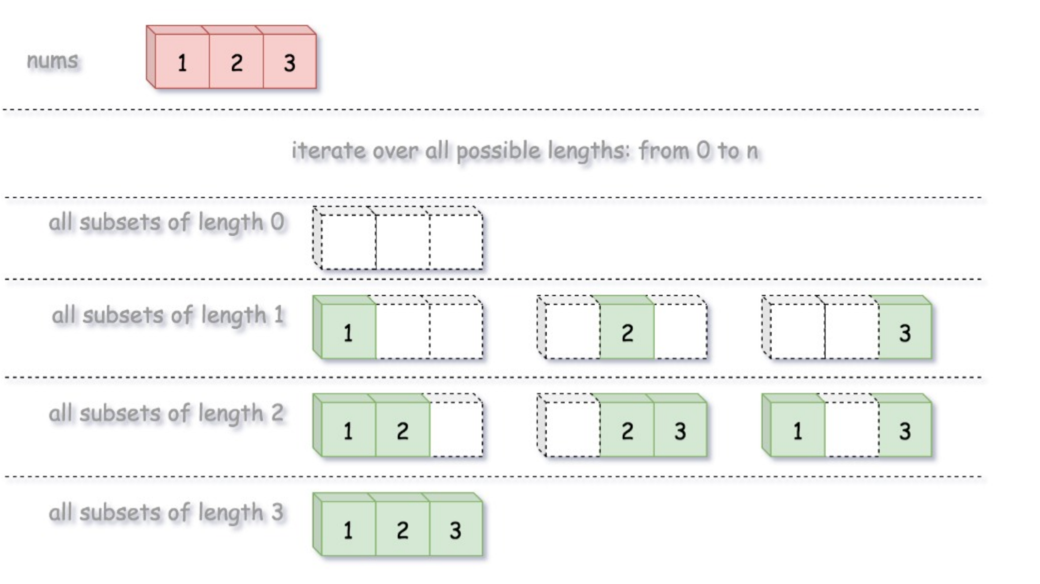
Approach 2: Backtracking

Algorithm

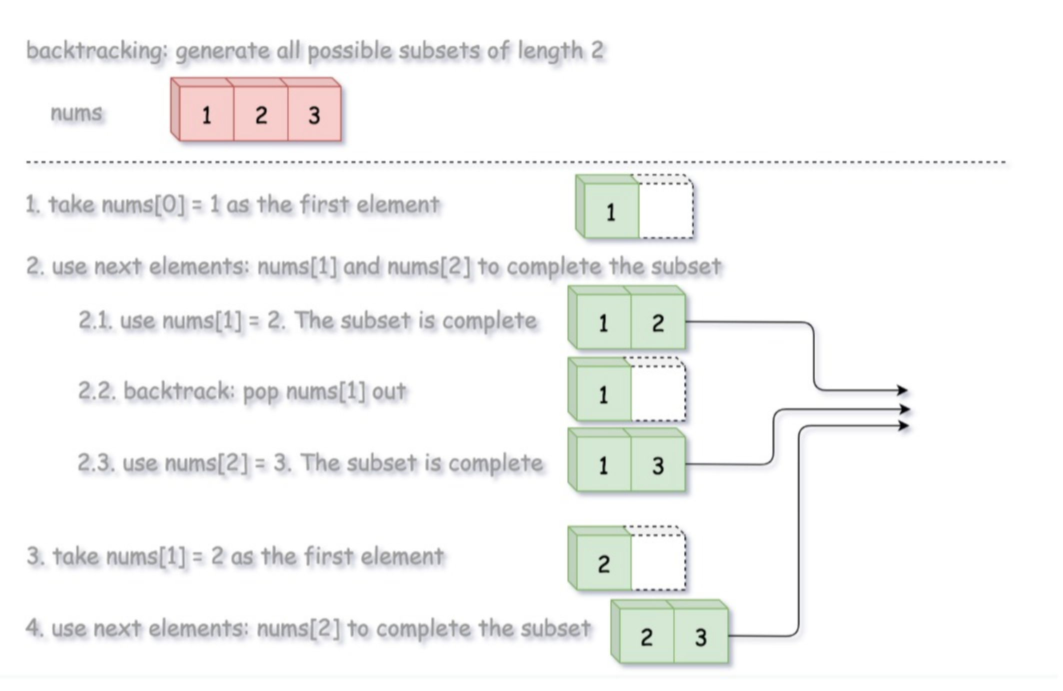
> Power set is all possible combinations of all possible *lengths*, from 0 to n.

Given the definition, the problem can also be interpreted as finding the *power set* from a sequence.

So, this time let us loop over *the length of combination*, rather than the candidate numbers, and generate all combinations for a given length with the help of *backtracking* technique.



> 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.



Algorithm

We define a backtrack function named `backtrack(first, curr)` which takes the index of first element to add and a current combination as arguments.

- If the current combination is done, we add the combination to the final output.
- Otherwise, we iterate over the indexes `i` from `first` to the length of the entire sequence `n`.
  - Add integer `nums[i]` into the current combination `curr`.
  - Proceed to add more integers into the combination : `backtrack(i + 1, curr)`.
  - Backtrack by removing `nums[i]` from `curr`.

Implementation

Java Python Copy

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def backtrack(first = 0, curr = []):
            # if the combination is done
            if len(curr) == k:
                output.append(curr[:])
            for i in range(first, n):
                # add nums[i] into the current combination
                curr.append(nums[i])
                # use next integers to complete the combination
                backtrack(i + 1, curr)
                # backtrack
                curr.pop()

        output = []
        n = len(nums)
        for k in range(n + 1):
            backtrack()
        return output
```

Complexity Analysis

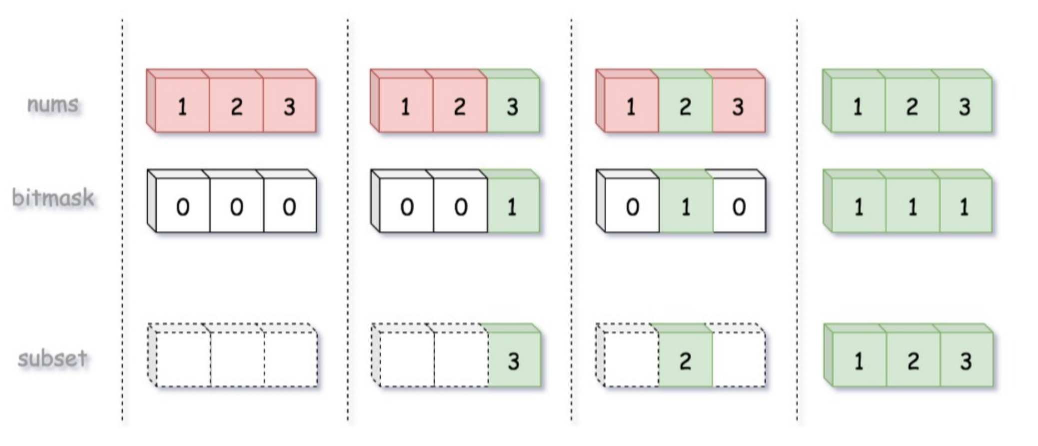
- Time complexity: $\mathcal{O}(N \times 2^N)$ to generate all subsets and then copy them into output list.
- Space complexity: $\mathcal{O}(N \times 2^N)$ to keep all the subsets of length N, since each of N elements could be present or absent.

Approach 3: Lexicographic (Binary Sorted) Subsets

Intuition

The idea of this solution is originated from Donald E. Knuth.

> The idea is that we map each subset to a bitmask of length n, where `1` on the *ith* position in bitmask means the presence of `nums[i]` in the subset, and `0` means its absence.



For instance, the bitmask `0..00` (all zeros) corresponds to an empty subset, and the bitmask `1..11` (all ones) corresponds to the entire input array `nums`.

Hence to solve the initial problem, we just need to generate n bitmasks from `0..00` to `1..11`.

It might seem simple at first glance to generate binary numbers, but the real problem here is how to deal with zero left padding, because one has to generate bitmasks of fixed length, *i.e.* `001` and not just `1`. For that one could use standard bit manipulation trick:

Java Python Copy

```
nth_bit = 1 << n
for i in range(2**n):
    # generate bitmask, from 0..00 to 1..11
    bitmask = bin(i | nth_bit)[3:]
```

or keep it simple stupid and shift iteration limits:

Java Python Copy

```
for i in range(2**n, 2**(n + 1)):
    # generate bitmask, from 0..00 to 1..11
    bitmask = bin(i)[3:]
```

Algorithm

- Generate all possible binary bitmasks of length n.
- Map a subset to each bitmask: `1` on the *ith* position in bitmask means the presence of `nums[i]` in the subset, and `0` means its absence.
- Return output list.

Implementation

Java Python Copy

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        output = []

        for i in range(2**n, 2**(n + 1)):
            # generate bitmask, from 0..00 to 1..11
            bitmask = bin(i)[3:]

            # append subset corresponding to that bitmask
            output.append([nums[j] for j in range(n) if bitmask[j] == '1'])

        return output
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N \times 2^N)$ to generate all subsets and then copy them into output list.
- Space complexity: $\mathcal{O}(N \times 2^N)$ to keep all the subsets of length N, since each of N elements could be present or absent.

Rate this article:

Previous Next

Comments: 43 Sort By

Type comment here... (Markdown is supported)

Preview Post

KP1975 51 December 31, 2019 8:42 AM

back track solution doing unnecessary computation. return statement missing.

```
    if (curr.size() == k) {
        output.add(new ArrayList(curr));
        return;   //missing
```

Read More

49 Share Reply

SHOW 3 REPLIES

ugurthesolver 105 February 14, 2020 4:00 AM

It says the approach 1 is recursion but it's not a recursive solution. It's more like a dynamic problem approach.
Please fix it so people wouldn't misunderstand it

37 Share Reply

silentcoder9 59 January 8, 2020 5:41 AM

Perhaps it could be made more simple by adding each new decision to the previously made choices.

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
```

Read More

14 Share Reply

SHOW 3 REPLIES

fudonglai 964 February 2, 2020 6:15 PM

I think the space complexity is O(N*2^N). I know there are 2^N subsets for the length of N, but every subset needs O(N) to store. So the total space should be O(N*2^N). Am I wrong?

10 Share Reply

rjsr 8 January 7, 2020 4:56 AM

How is approach 1 O(N 2^N). Even though the outer loop runs n times, the inner loop is not always O(2^N). You end up running a total of (2^0 + 2^1 + .. 2^(N-1)) = 2^N commands. The time complexity should therefore be O(2^N).

8 Share Reply

SHOW 2 REPLIES

stevphan 6 December 30, 2019 1:36 AM

I see different backtracking problems doing either a "swap" (such as permutations) or a "add and then remove" (such as this problem). Is there a way to easily distinguish when to use each way?

5 Share Reply

SHOW 2 REPLIES

AminiCK 554 February 7, 2020 3:36 PM

I feel like Approach 1 is actually a DP(Bottom Up) version.

7 Share Reply

Safadurimo 5 January 7, 2020 3:18 AM

There is no need to transform the bitsmask to a string: One can check directly on the loop counter if the j-th bit is set. Also the chosen loop intervall makes the reasoning harder: For n=3, simple interpret 000 as 0 and loop from 0 to 111(binary). I found this tutorial really helpful:
https://www.topcoder.com/community/competitive-programming/tutorials/a-bit-of-fun-fun-with-bits/

4 Share Reply

adriansky 105 March 18, 2020 2:53 AM

Why O(N*2^N) and not O(2^N)?

3 Share Reply

SHOW 1 REPLY

zoellin1220 1 June 12, 2020 6:42 AM

why is there a [3:] in the third method to generate bitmask?

1 Share Reply

SHOW 1 REPLY