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](https://en.wikipedia.org/wiki/Permutation#k-permutations_of_n): N!*N*!.
* [Combinations](https://en.wikipedia.org/wiki/Combination#Number_of_k-combinations): C\_N^k = \frac{N!}{(N - k)! k!}*CNk*​=(*N*−*k*)!*k*!*N*!​
* Subsets: 2^N2*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.

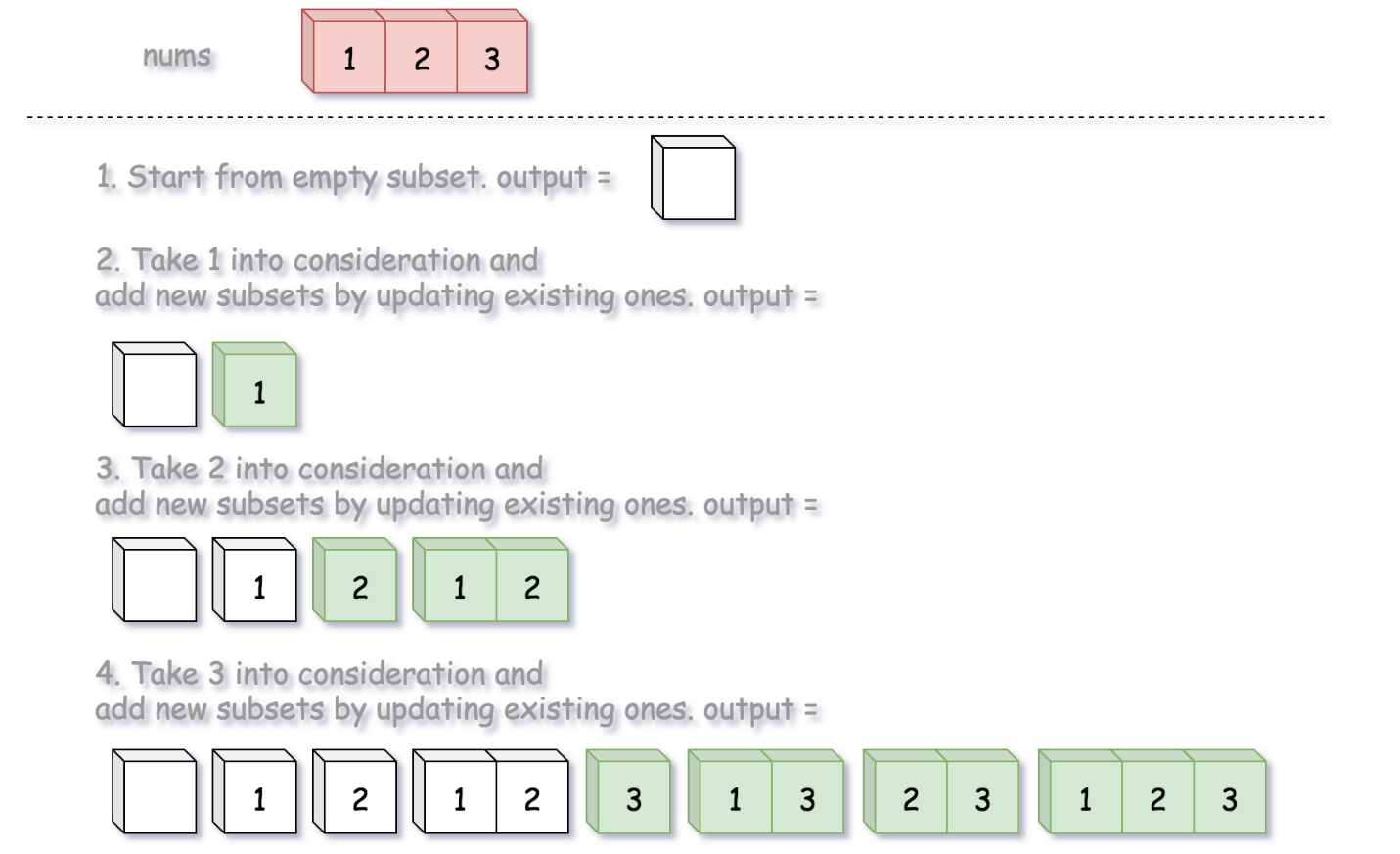
As one would see later, the third method could be a good candidate for the interview because it simplifies the problem to the generation of binary numbers, therefore it is easy to implement and verify that no solution is missing.

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

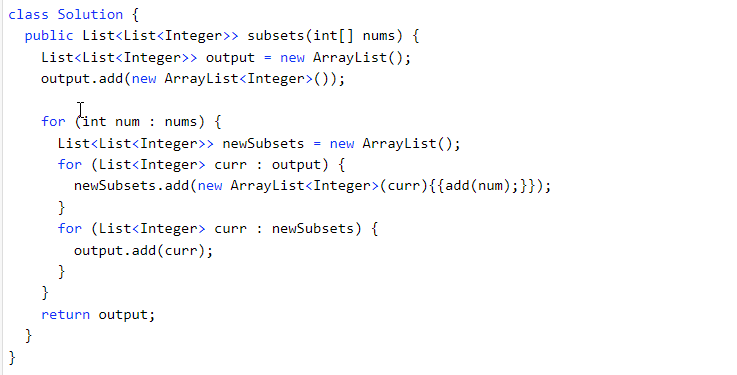
Approach 1: Cascading

**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**



**Complexity Analysis**

* Time complexity: O(*N*×2^N) to generate all subsets and then copy them into output list.
* Space complexity: O(*N*×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).

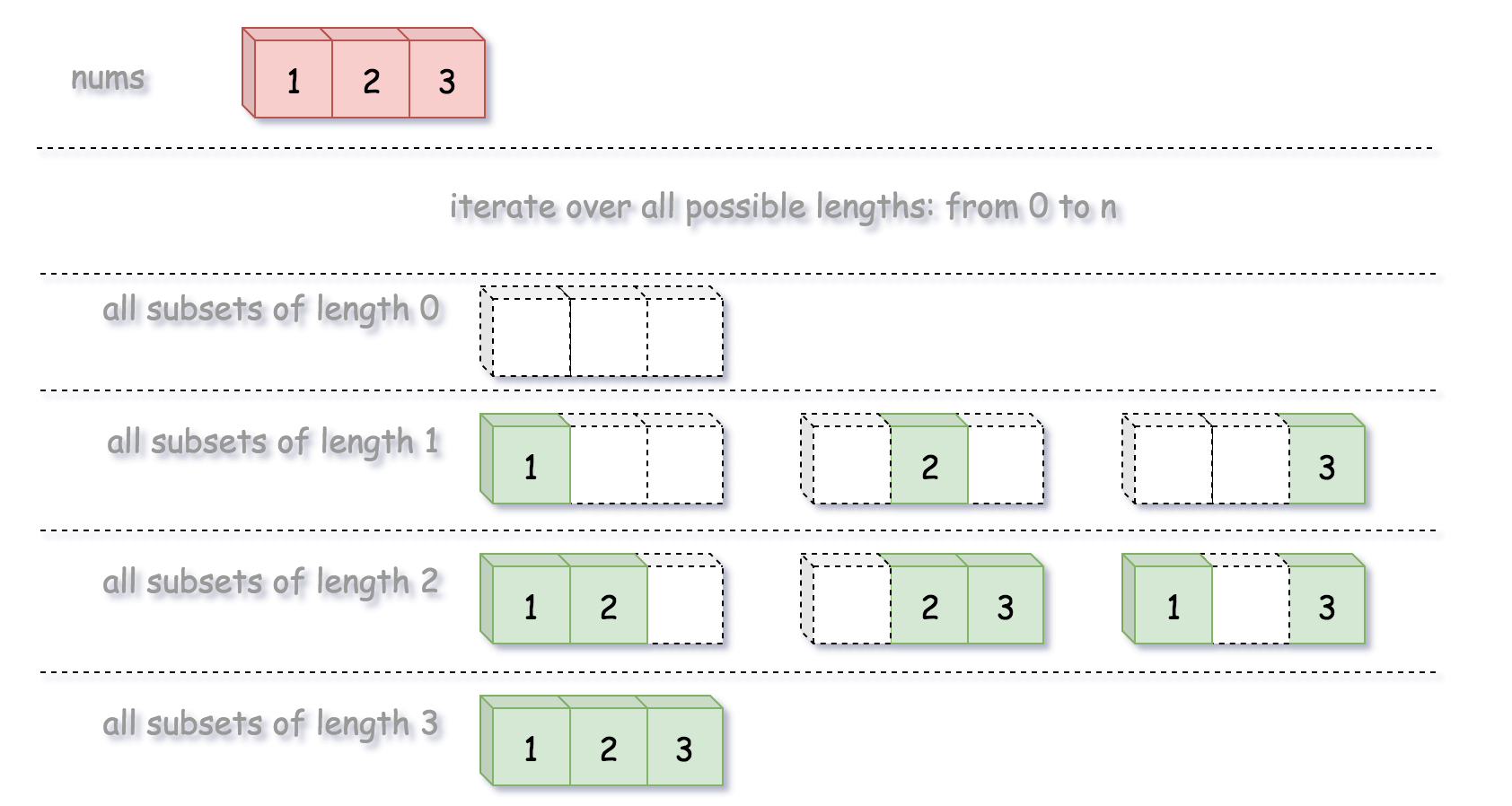
Approach 2: Backtracking

**Algorithm**

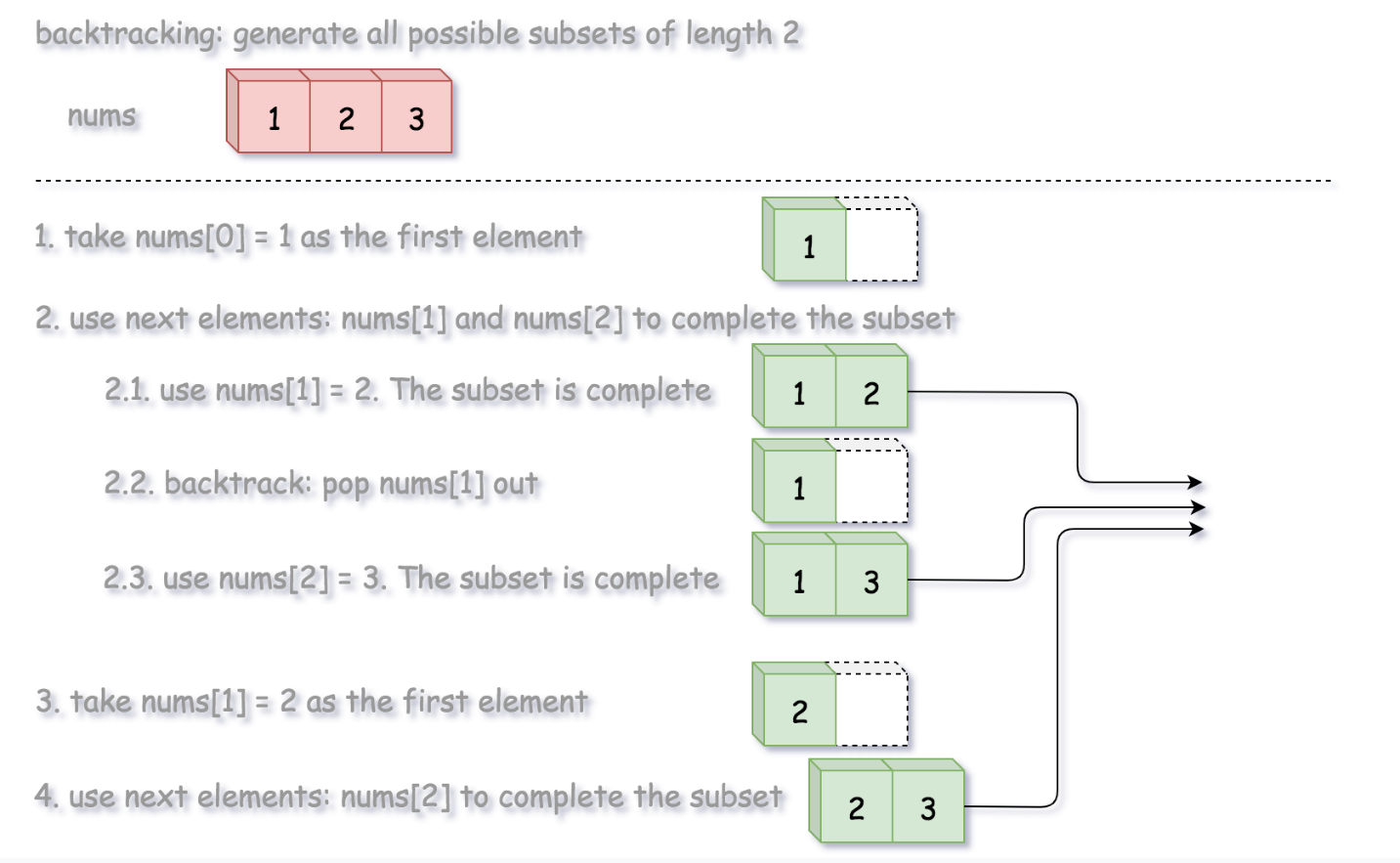
Power set is all possible combinations of all possible *lengths*, from 0 to n.

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](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2654/) 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.

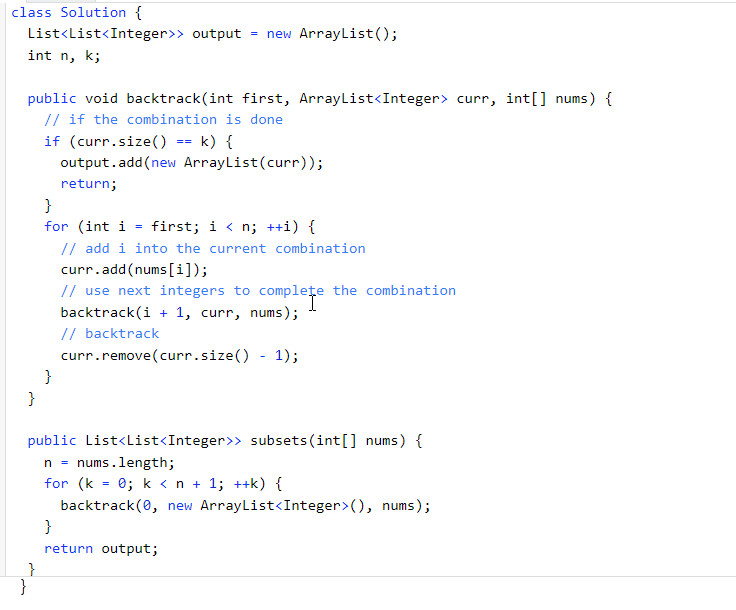


**Algorithm**

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**



**Complexity Analysis**

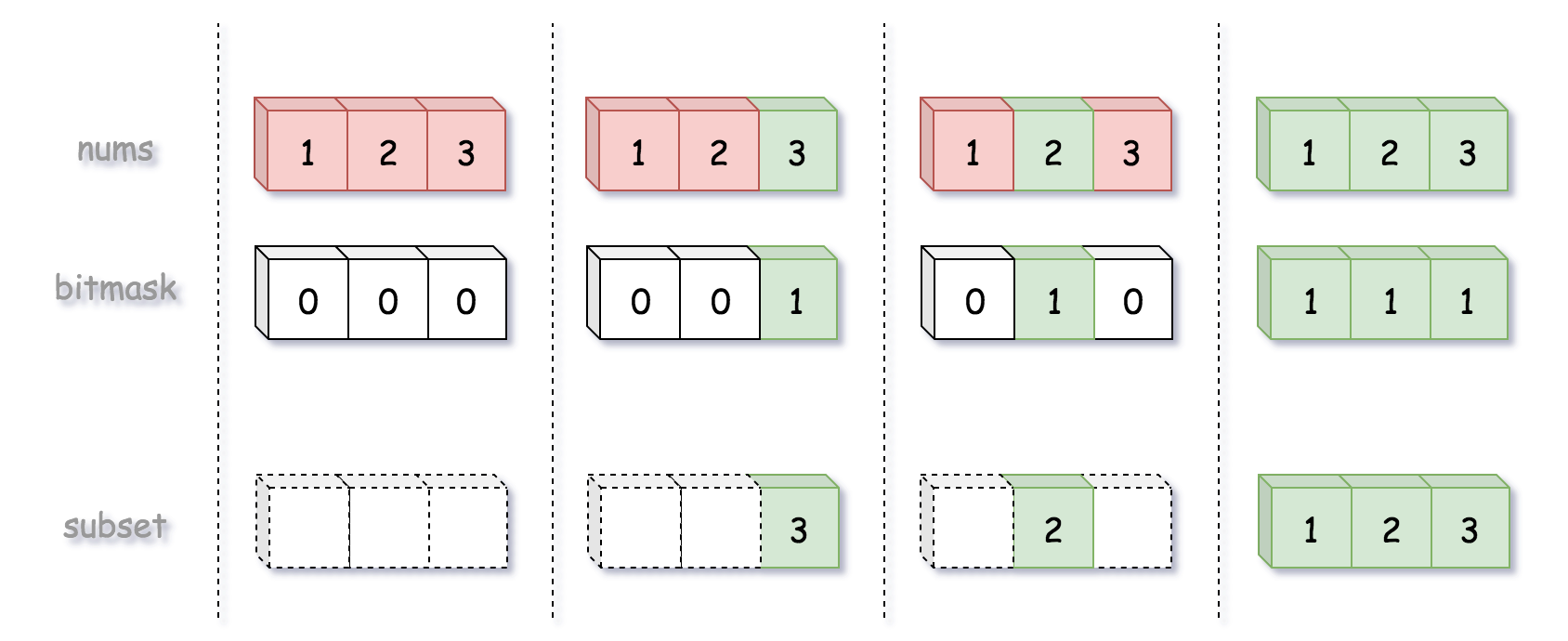
* Time complexity O(*N*×2^N) to generate all subsets and then copy them into output list.
* Space complexity:  O(*N*). We are using O(N) space to maintain curr, and are modifying curr in-place with backtracking. Note that for space complexity analysis, we do not count space that is *only* used for the purpose of returning output, so the output array is ignored.

Approach 3: Lexicographic (Binary Sorted) Subsets

**Intuition**

The idea of this solution is originated from [Donald E. Knuth](https://www-cs-faculty.stanford.edu/~knuth/taocp.html).

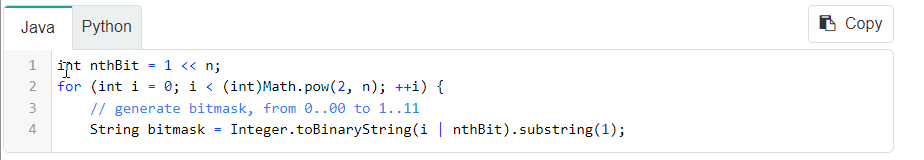
The idea is that we map each subset to a bitmask of length n, where 1 on the i*th* position in bitmask means the presence of nums[i] in the subset, and 0 means its absence.



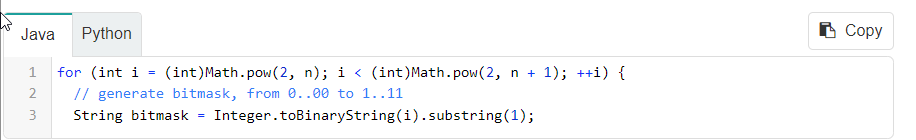
For instance, the bitmask 0..00 (all zeros) corresponds to an empty subset, and the bitmask 1..11 (all ones) corresponds to the entire input array nums.

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](https://en.wikipedia.org/wiki/Padding_(cryptography)#Zero_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:



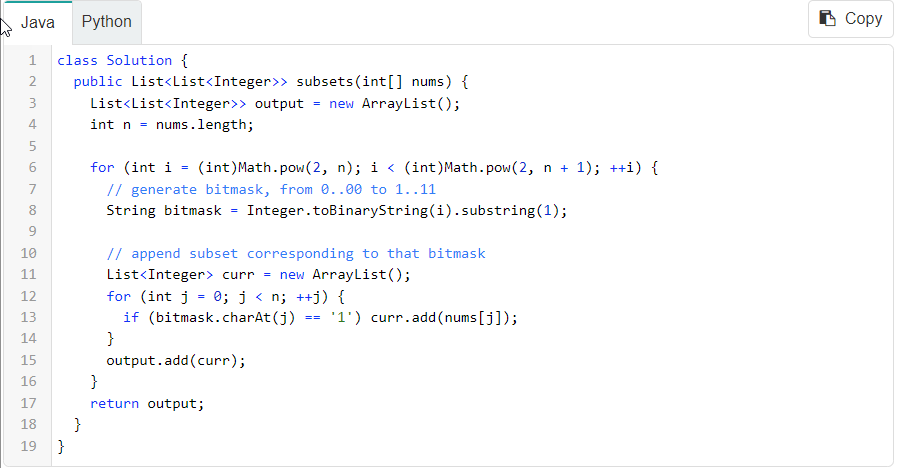
or keep it simple stupid and shift iteration limits:



**Algorithm**

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

**Implementation**



**Complexity Analysis**

* Time complexity:  O(N\*2^N) to generate all subsets and then copy them into output list.
* Space complexity: O(N \* 2^N) to keep all the subsets of length N, since each of Nelements could be present or absent.