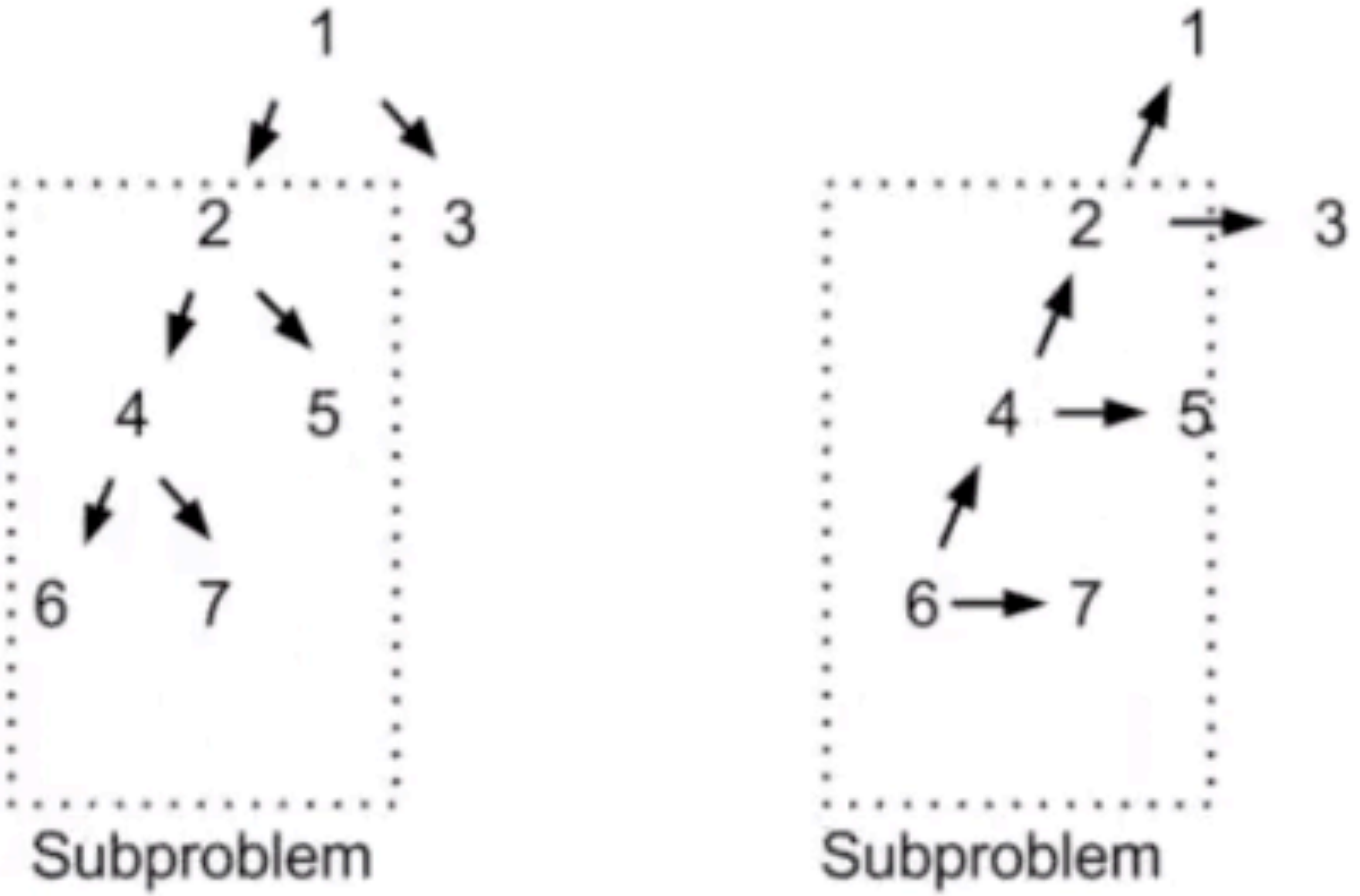
Q1.3 Reverse a binary tree upside down

Given a binary tree where all the right nodes are leaf nodes, flip it upside down and turn it into a tree with left leaf nodes. **For example**, turn these:



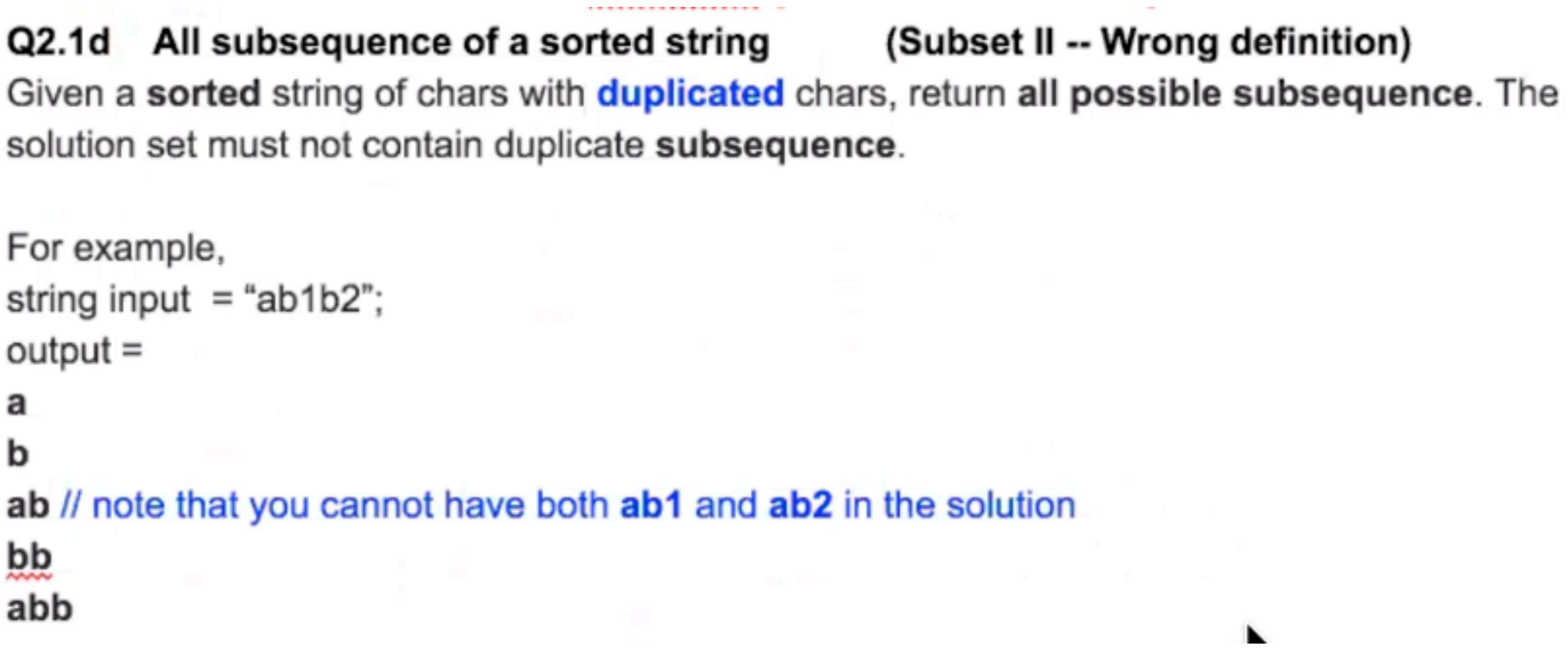
除了subproblem，我们在当前层需要做什么？

(1) root ->lChild->lChild = root->rChild

(2) root->lChild->rChild = root

(3) root->lChild = NULL

(4) root->rChild = NULL



Q2.1d All subsequence of a sorted string (Subset II -- Wrong definition)

Given a **sorted** string of chars with **duplicated** chars, return **all possible subsequence**. The solution set must not contain duplicate **subsequence**.

For example,

string input = “ab1b2”;

output =

a

b

ab // note that you cannot have both **ab1** and **ab2** in the solution

bb

abb

For example,

string input = "ab1b2";

output =

a

b

ab // note that you cannot have both **ab1** and **ab2** in the solution

bb

abb

a x b x c

0 0 0 empty

...

1 1 1 full-set

string = "a b1 b2 b3 c"

a 3xb c

0 x 0 x 0

1 1 1

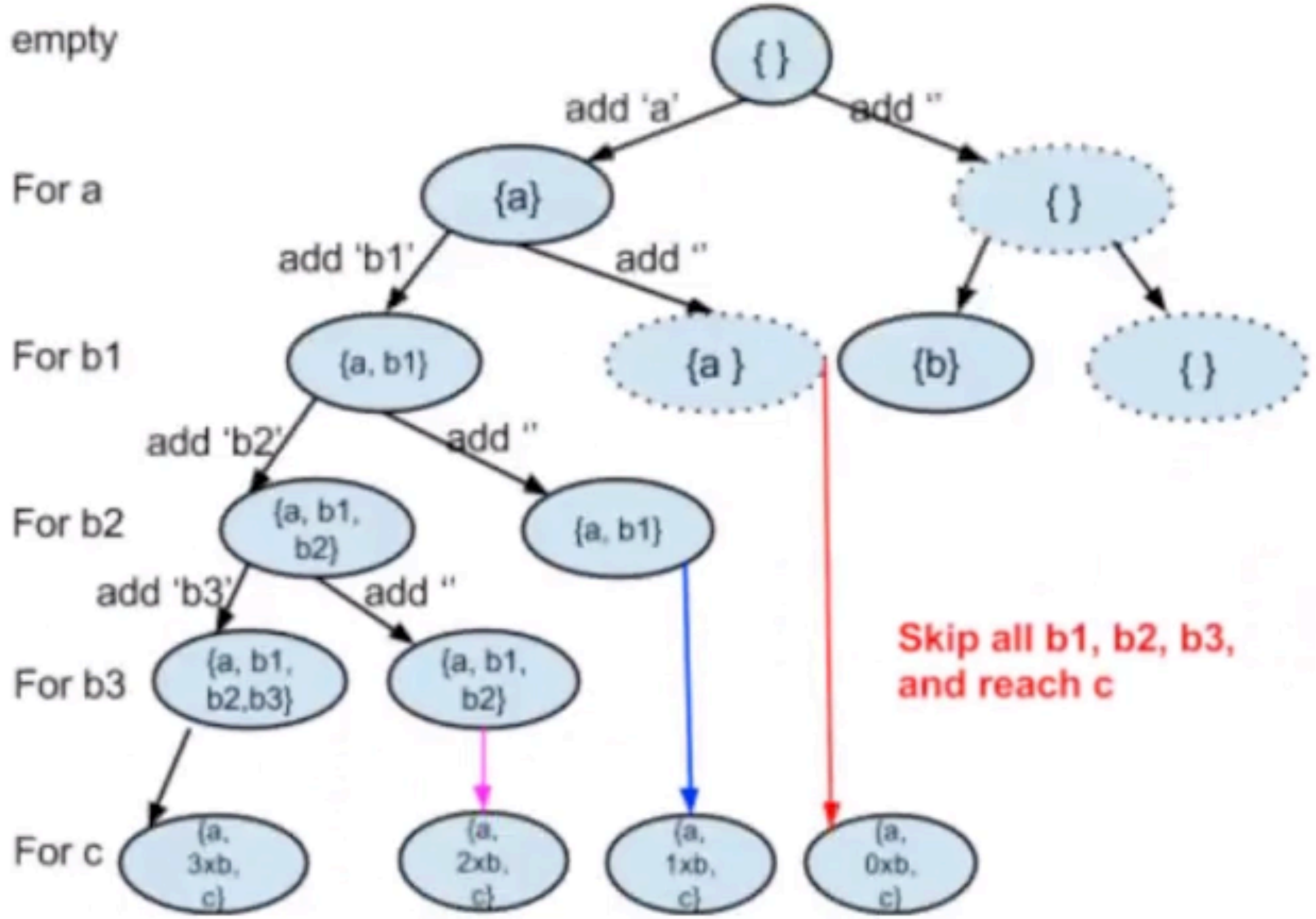
2

3

DFS 基本方法：

1 what does it store on each level? (每层代表什么意义？一般来讲解题之前就知道DFS要recurse多少层)　　5 elements → 5 level, each level decides whether or not add this current element

2 How many different states should we try to put on this level? （每层有多少个状态/case需要try？）



Skip all b1, b2, b3, and reach c

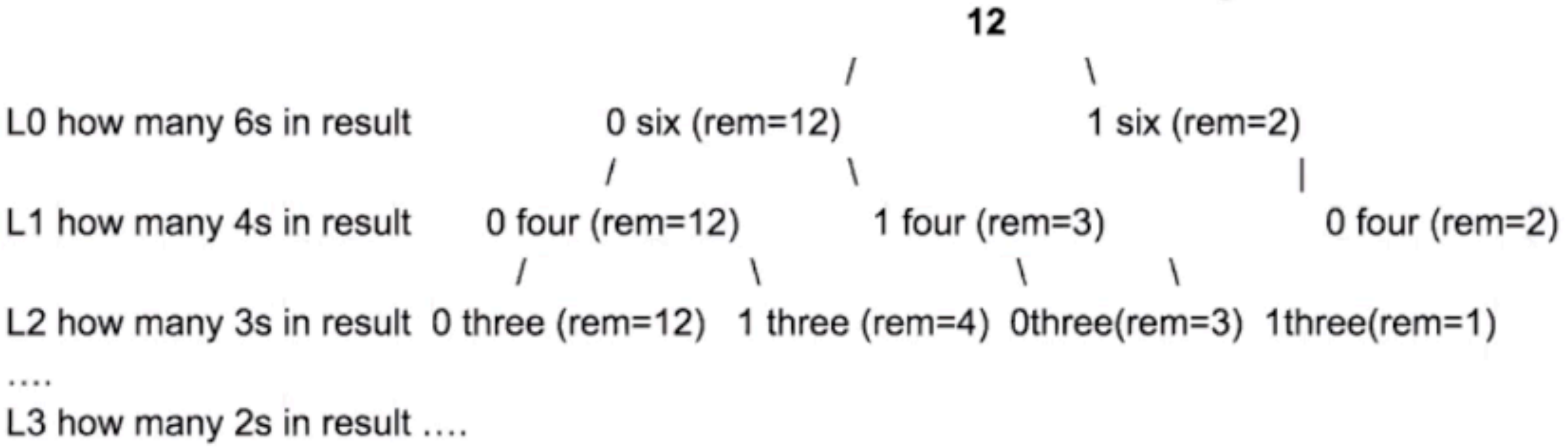
```
private void helper(List<Integer> solution, int[] input, int index){
       if(index == input.length){
           print solution;    // base case
           return;
       }
       // Case1: Add num[index]
       solution.add(input[index]);                  // +b
       helper(solution, input, index + 1);
       solution.remove(tmp.size() - 1);

       // Skips all the rest of duplicated letters (e.g. b1 b2
       // b3…. in this example)
       while(index < input.length - 1 &&
             input[index + 1] == input[index]) {
           index++;
       }
```

```
       // Case2: Do not add num[index]
       helper(solution, input, index + 1);
   }
```

Q2.3b Print all valid combination of factors that form an integer.

12
= 6 x 2
= 4 x 3
= 3 x 2 x 2
...

1. **what does it store on each level? (**每层代表什么意义？一般来讲解题之前就知道DFS要recurse多少层) 6 4 3 2 → 4 levels
2. **How many different states should we try to put on this level?** （每层有多少个状态/case 需要try？）



12

L0 how many 6s in result: 0 six (rem=12), 1 six (rem=2)

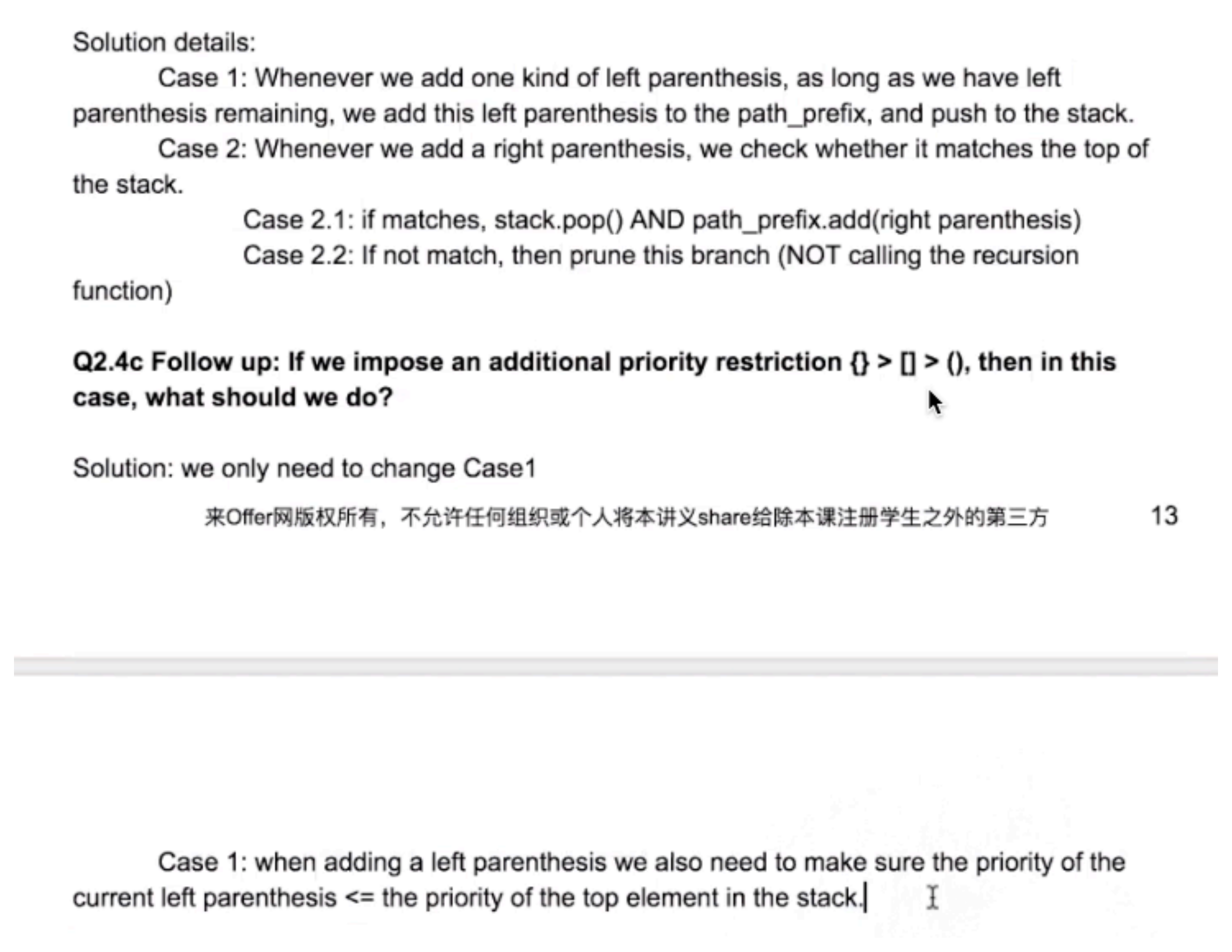
L1 how many 4s in result: 0 four (rem=12), 1 four (rem=3), 0 four (rem=2)

L2 how many 3s in result: 0 three (rem=12), 1 three (rem=4), 0three(rem=3), 1three(rem=1)

....

L3 how many 2s in result ....

Time = O((n/2)^4) = O((n/2) ^ factor)



Solution details:

Case 1: Whenever we add one kind of left parenthesis, as long as we have left parenthesis remaining, we add this left parenthesis to the path_prefix, and push to the stack.

Case 2: Whenever we add a right parenthesis, we check whether it matches the top of the stack.

Case 2.1: if matches, stack.pop() AND path_prefix.add(right parenthesis)

Case 2.2: If not match, then prune this branch (NOT calling the recursion function)

Q2.4c Follow up: If we impose an additional priority restriction {} > [] > (), then in this case, what should we do?

Solution: we only need to change Case1

Case 1: when adding a left parenthesis we also need to make sure the priority of the current left parenthesis <= the priority of the top element in the stack.

Q3: 2,3,4-SUM questions

Find X-elements from a (**unsorted / sorted**) array such that their sum is equal to a target value.

Q3.1 2SUM

Given an array , how to find two numbers in it such that their sum is equal to a target number.

Solution 1: (naive way)

```
for i {
        for j {
                check if a[i] + a[j] == target
        }
}
```

Assumptions:

- sorted/unsorted
- duplicate
- array size
- output index or true/false
- return one or many
- int + int overflow?

Assumptions:

- sorted/unsorted
- duplicate
- array size
- output index or true/false
- return one or many
- int + int overflow?
- Optimize time or space?

What if the array is unsorted? And we need to return the indices of any one solution.

Solution 2:
hash_map<key = value, value = its index>

```
Iterate over the whole array, for the current index i:
    check whether (target - input[i]) is in the hash_map or not.
        If yes, return hash_map.get(target - input[i]) and i.
```

If sorted, use two pointers

```
xxxxxxXxxxxxxxxxxxxxxxxxxxxxxxYxxxxxxx
      i →                  ← j
```

Case 1: if input[i] + input[j] < target; i++

Case 2: if input[i] + input[j] > target; j++

Case 3: if input[i] + input[j] == target; return i and j.

Follow up: what if the memory in one machine is < the size of the array (assuming sorted).

1TB data on disk vs 1GB memory

```
xxxxxxx               xxxxxxx  xxxxxxxxxxxxxxxxxxxxxxxxxxxx          xxxxxxx
[0...250MB]                                                        last 250MB
                       i→                                          ←j
```

Q3.2 3SUM

Solution 1: for for for → O(n^3)

Solution 2:

```
    for i {                    n
            run 2SUM()         → O(n)
    }
Time = O(n^2)
```

Q3.3 4SUM

Solution 1: for for for for → O(n^4)

Solution 2:

```
    for i {
            for j {
                    run 2SUM
            }
    }
Time = O(n^3)
```

```
// Method 3: HashMap O(n ^ 2).
public boolean existIII(int[] array, int target) {
  // Assumptions: array is not null, array.length >= 4.
  Map<Integer, Pair> map = new HashMap<>();
  // the order of traversing i, j is not arbitrary, we should guarantee
  // we can always look at the pair with the smallest right index.
  for (int j = 1; j < array.length; j++) {// j is the right index of the pair
    for (int i = 0; i < j; i++) {            // i is the left index of the pair, j < i
      int pairSum = array[i] + array[j];
```

```
      // we need to guarantee there exists another pair with right index
      // smaller than the current pair's left index.
      if (map.containsKey(target - pairSum) && map.get(target - pairSum).right < i) {
        return true;
      }
      // we only need to store the pair with smallest right index.
      if (!map.containsKey(pairSum)) {
        map.put(pairSum, new Pair(i, j));
      }
    }
  }
  return false;
```