Recursion

Tilo.Dickopp@gmail.com

continue @ ~~10:40~~
~~12:25~~
~~1:38pm~~
2:37pm

② Combinatorial Enumeration

- „Find all $\boxed{\text{variations}}$ of some input."
- Brut force
- Often exponentially many variations

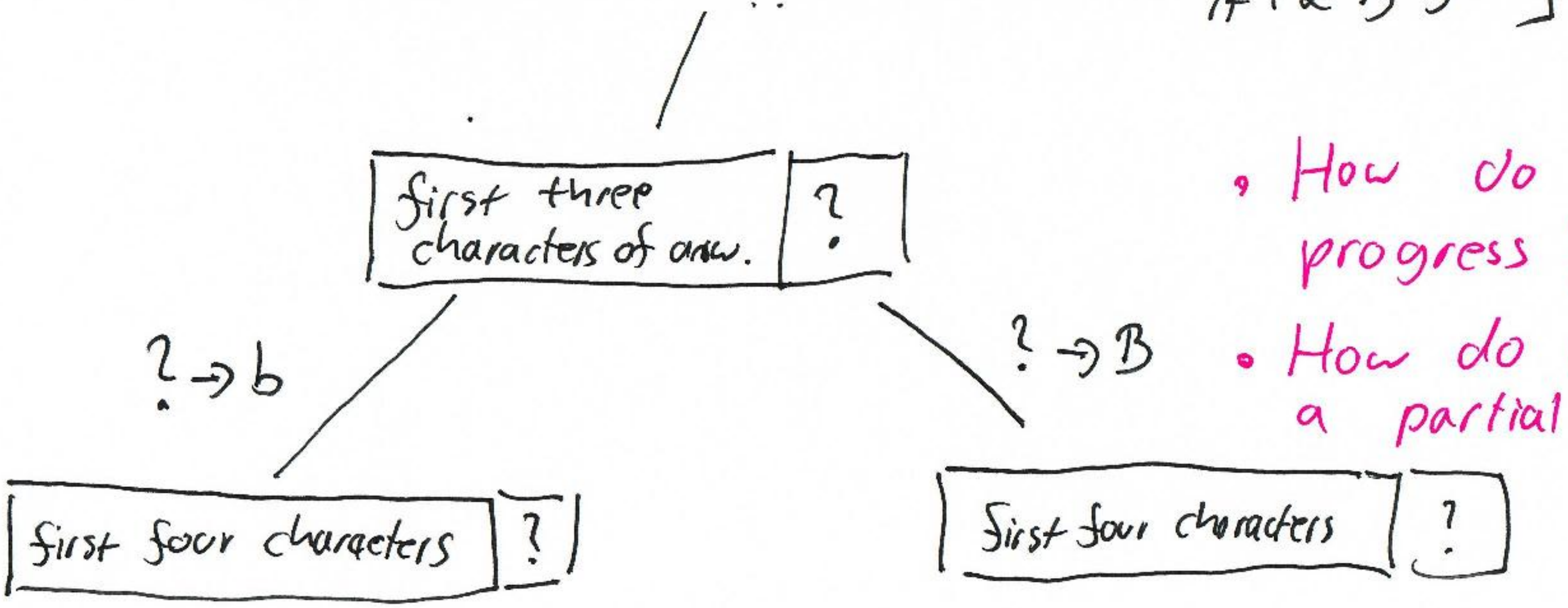
Outline of solution for „Combinatorial ~~Enumeratio~~ Enumeration" questions:

```
def recursiveProcedure(original ~~input~~ input, the progress I've made,
                                          collecting overall result)
    check if I can backtrack?
         (can I stop early?)
    check for base case: am I done?
    modify shared state
    recursive calls
    un-modify shared state
```

③ Letter Case Permutations

Given an input string, return all variations of this string ~~wheer~~ where letters are upper and lower-case.

Example: input = "a12b3" output = ["a12b3",
"A12b3",
"a12B3",
"A12B3"]



- How do I track progress?
- How do I store a partial solution (slate)?

④ Game Plan/Structure

① Chit-Chat / questions about you

② Find a solution

- understanding the question
- ask for example
- what is in scope (edge cases)
- don't ask question just for the sake of asking questions
- find a brute-force solution
- ask: „can I do better?" ⇒ Space & Time complexity
- explain ideas in English not in (pseudo) code

50% of time for „medium questions

③ Code the solution

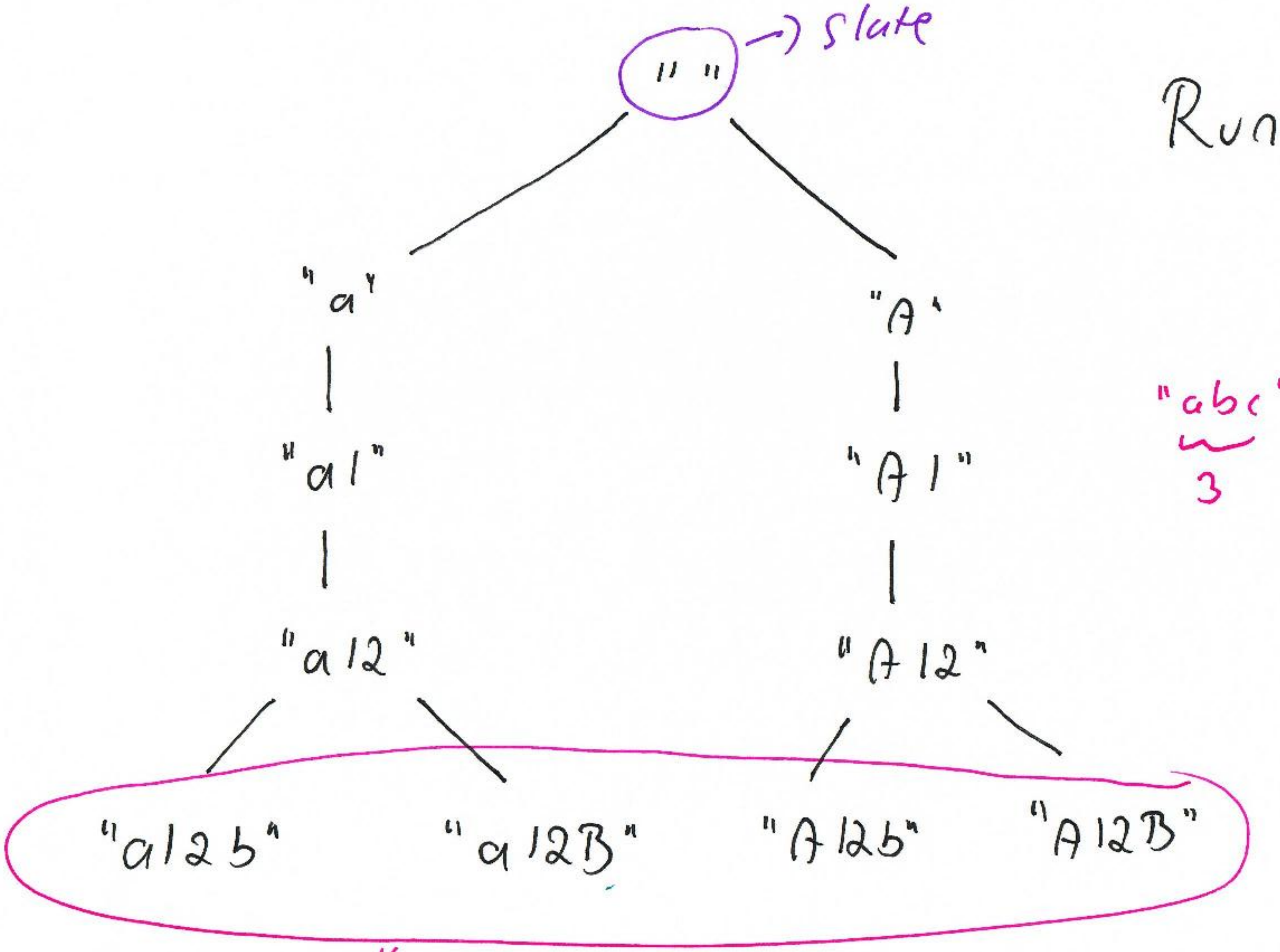
- write code
- don't worry about new ideas ⇒ to do list
- walk through the solution
  - ⇒ show example
  - ⇒ catch typical errors

50% of time

④ Ending interview

⑤ Letter Case Permutations

input = "a12b~~3~~"



Runtime? $O(2^N)$

$\underbrace{\text{"abc"}}_{3} \Rightarrow$ "abc"
"Abc"
"ABc"
"AbC"
"ABC"
"aBc"
"aBC"
"abC"

$2^3 = 8$

⇓

if for every recursive call we do constant work, the runtime can be found by counting the results

⑥ Letter case Permutations: code

```
List<String> letterCasePermutations(String input) {
    var results = new ArrayList<String>();
    helper(input.toCharArray(), 0, new char[input.length()], results);
    return results;
}

void helper(char[] input, int pos, char[] slate, List<String> results)
    if( pos >= input.length) {
        results.add( new String(slate));
        return;                 ↳ copy?
    }
    if ( isLetter(input[pos])) {
        ~~input~~ slate[pos] = toLower(input[pos]);
        helper(input, pos+1, slate, results);
        slate[pos] = toUpper(input[pos]);
        helper(input, pos+1, slate, results);   → un-modify?
    } else {
        slate[pos] = input[pos];
        helper(input, pos+1, slate, results); }}
```

Memory:

$O(N+N+N\cdot 2^N) =$

$O(N\cdot 2^N)$

(7) Subsets

Given a list of distinct objects (e.g. numbers) return all „subsets".

Example: input = [1, 2, 3]   output = [[1], [2], [3], [],
[1, 2], [1, 3], [2, 3],
[1, 2, 3]]

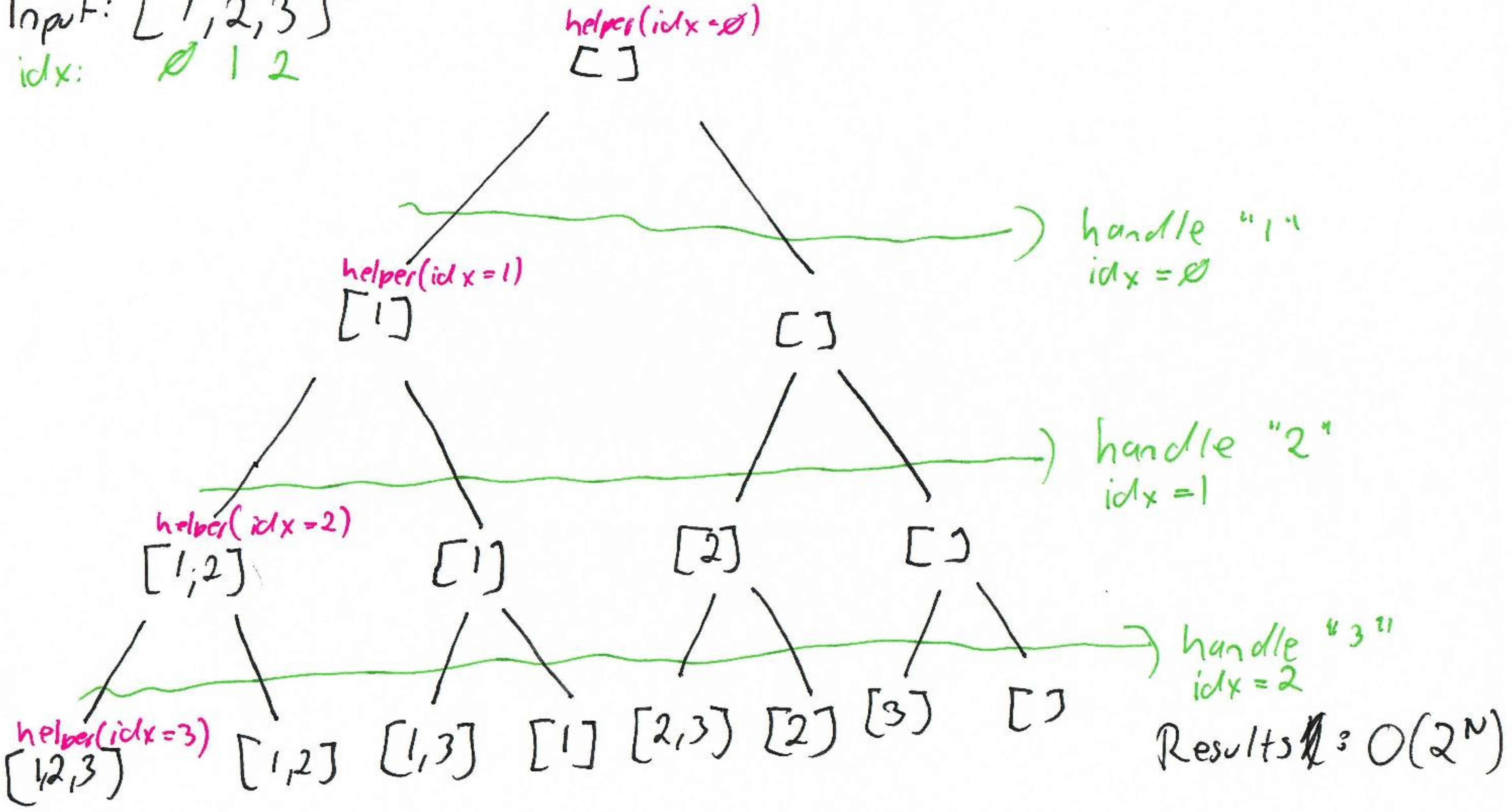
Memory: $O(2^N \cdot N)$ eg: $\underbrace{2^3}_{\text{number of subsets}} \cdot \underbrace{3}_{\text{upper bound of size of subset}}$

partial solution: [1] ⇒ could be: done
⇒ could be: begining of e.g. [1, 3], etc

idea: keep track of the numbers in input that we have already „used".

⑧ Subsets

Input: [1,2,3]
idx: ∅ 1 2



shared slate: ~~[ ]~~ ~~[3]~~ $\boxed{\text{~~[3,2]~~}}$ ~~[3,2,2]~~ . [3, 2, 2, 1]

results = [[ ], [3], [3,2], [3, 2, 2]

⑨ Subsets: Code

```
List<List<Integer>> subsets(List<~~Input~~ Integer> input){
    var results = new ArrayList<List<Integer>>();
    helper(input, 0, new ArrayList<Integer>(), results);   sort input to
    return results;                                          handle duplicates
}
void helper(List<Integer> input, int idx, List<Integer> slate,
            List<List<Integer>> results){
    if (idx >= input.size()){
        results.add(new ArrayList<>(slate));
        return;
    }
    slate.add(input.get(idx));
    helper(input, idx+1, slate, results);
    slate.remove(slate.size() -1); // un-modify: restore slate to what
                                        it was before
    helper(input, idx+1, slate, results);
}
```

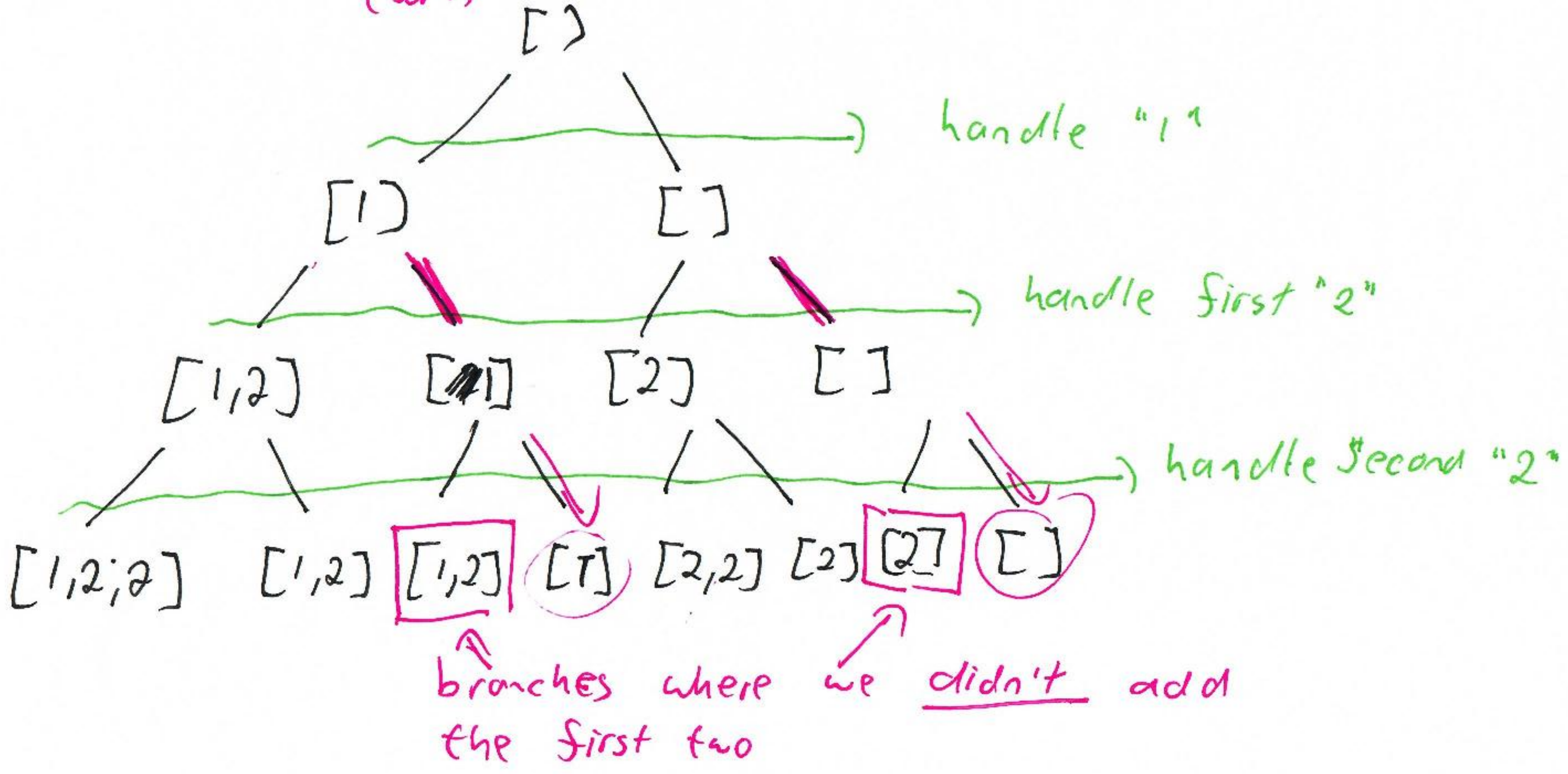
```
int nextDifferent(
  List<Integer> input, int idx)
  for(int i = idx++; i < input.
     size(); i++) {
    if(input.get(i) !=
       input.get(idx))
      return i; }}
  return input.size(); }
```

$\boxed{idx+1}$ → for duplicates do not just proceed to the next number (idx+1) but the next <u>different</u> number (idx + x)

⑩ Subsets with duplicates:

Example: Input = [1, 2, 2] Output = [[ ], [1], [2], ~~[2]~~, [1, 2], [2, 2], ~~[1, 2]~~, [1, 2, 2]]

make sure duplicates appear next to each other (sort)



11 All permutations

return all permutations of a given input:

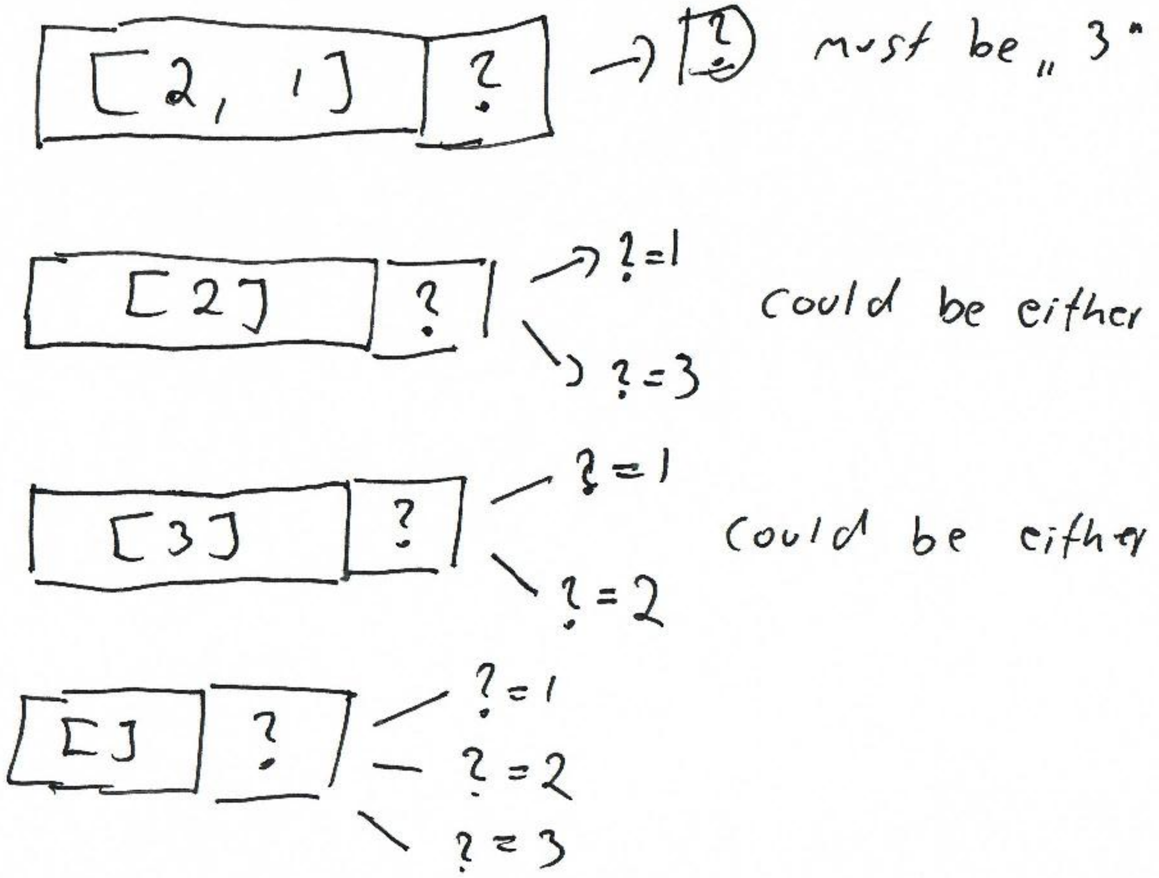
Example: input = [1, 2, 3]   output = [[1,2,3], [1,3,2],
[2,1,3], [2,3,1],
[3,1,2], [3,2,1]]

partial solutions: what next?

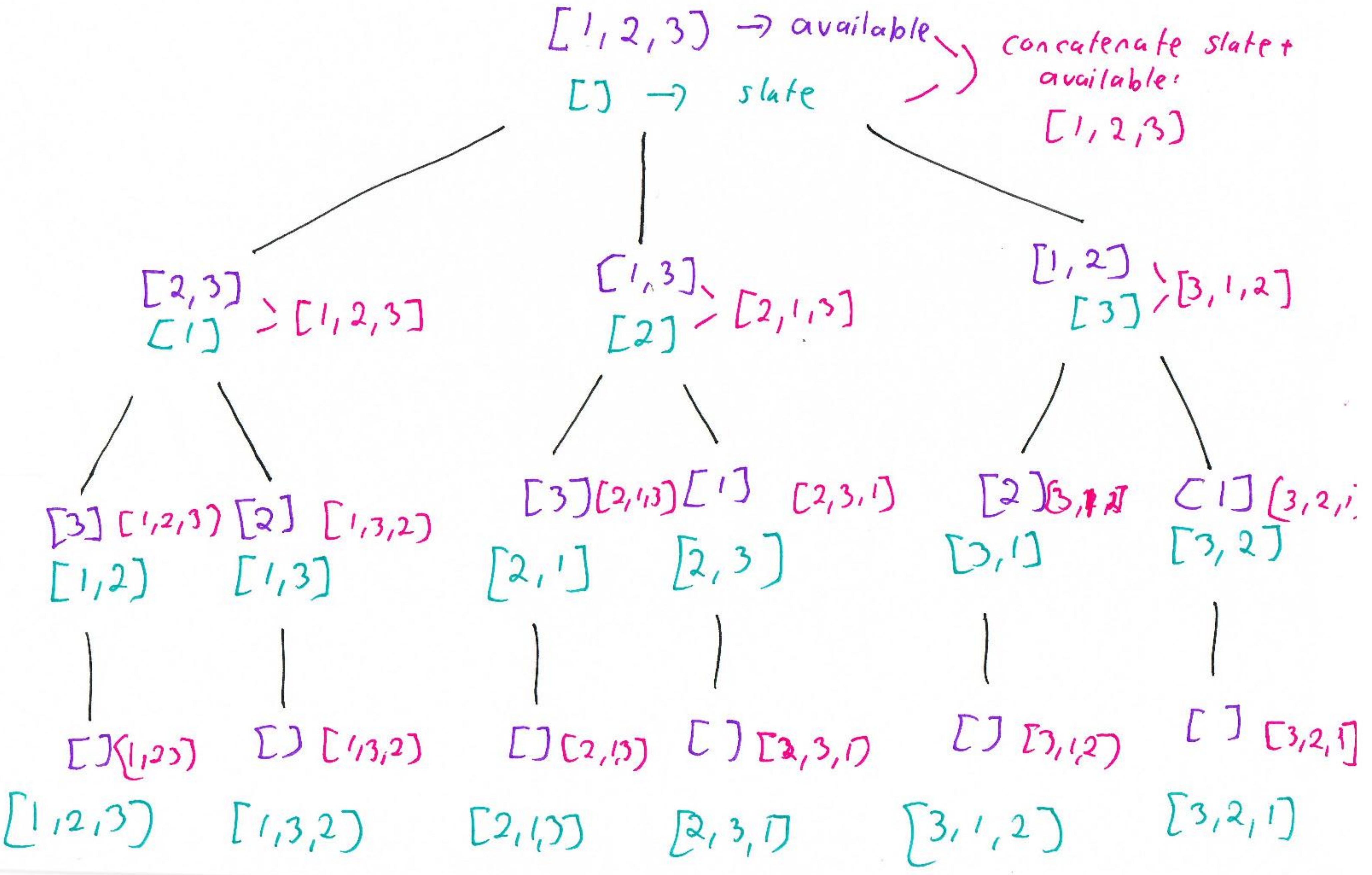
solution of length „2"

solution of length "1"

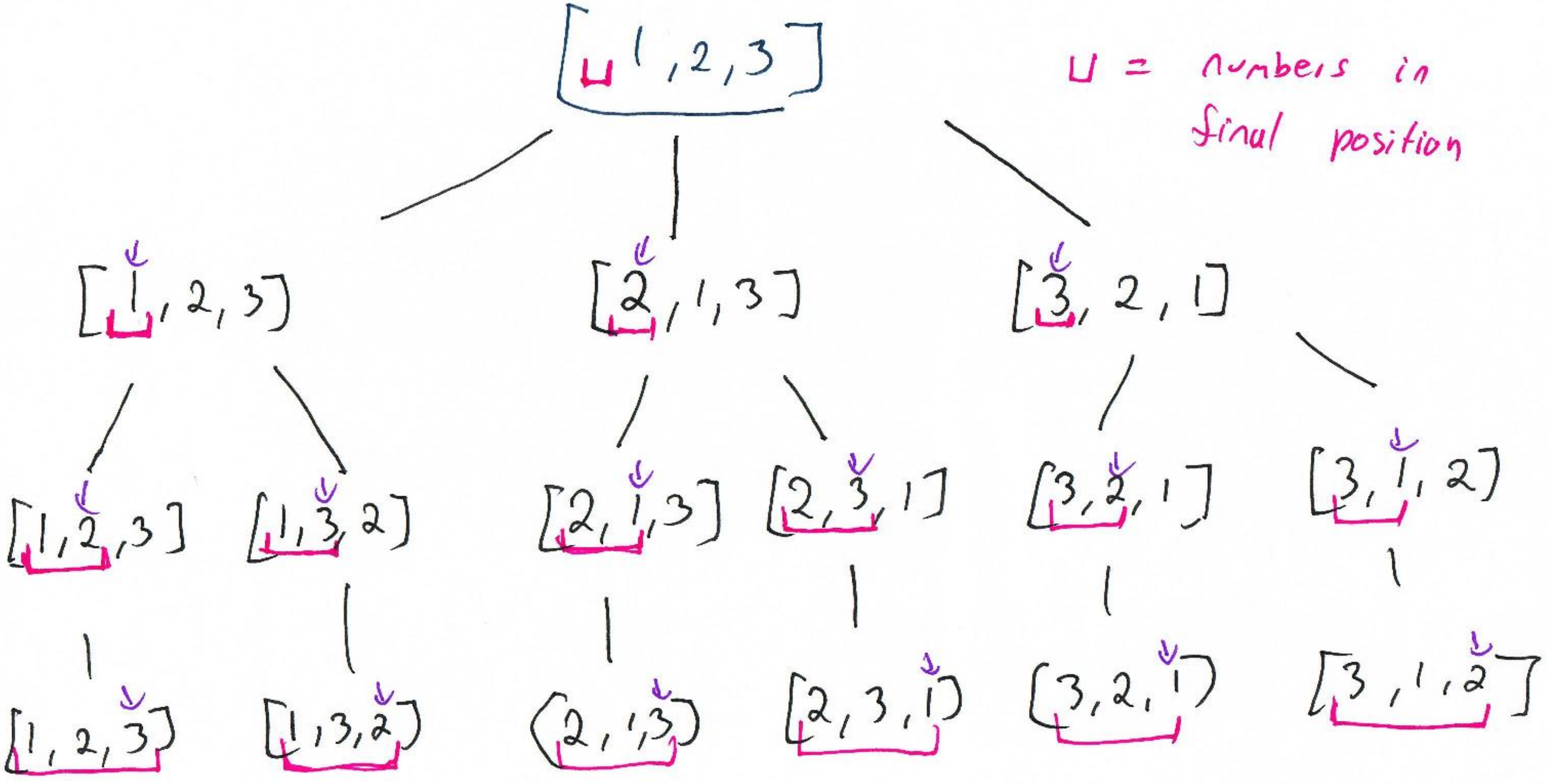
empty solution



(12) All permutations



(13) All permutations



Number of solutions: $O(N!)$

(14) All permutations : Code

```
List<List<Integer>> permutations(List<Integer> input) {
    var results = new ArrayList<List<Integer>>();
    helper(new ArrayList<>(input), ∅, results);
    return results;
}
void helper(List<Integer> slate, int placed, List<List<Integer>> results) {
    if (placed >= slate.size()) {
        results.add(new ArrayList<>(slate));
        return;
    }
    for (int i = placed; i < slate.size(); i++) {
        swap(slate, i, placed);
        helper(slate, placed+1, results);
        swap(slate, placed, i);
    }
}
```

int placed → number of elements in their final position

Memory: $O(N \cdot N!)$

Time: $O(N \cdot N!)$

True Memory:

$$O(\underset{\text{slate}}{N} + \underset{\text{call stack}}{N} + \underset{\text{results}}{N \cdot N!})$$

(15) Well formed parantheses

Given a number k of pairs of parantheses generate all well formed strings:

<u>Example:</u>

K = 3 output = ["()()()", "()(())", "(())()", "((()))", "(()())"]

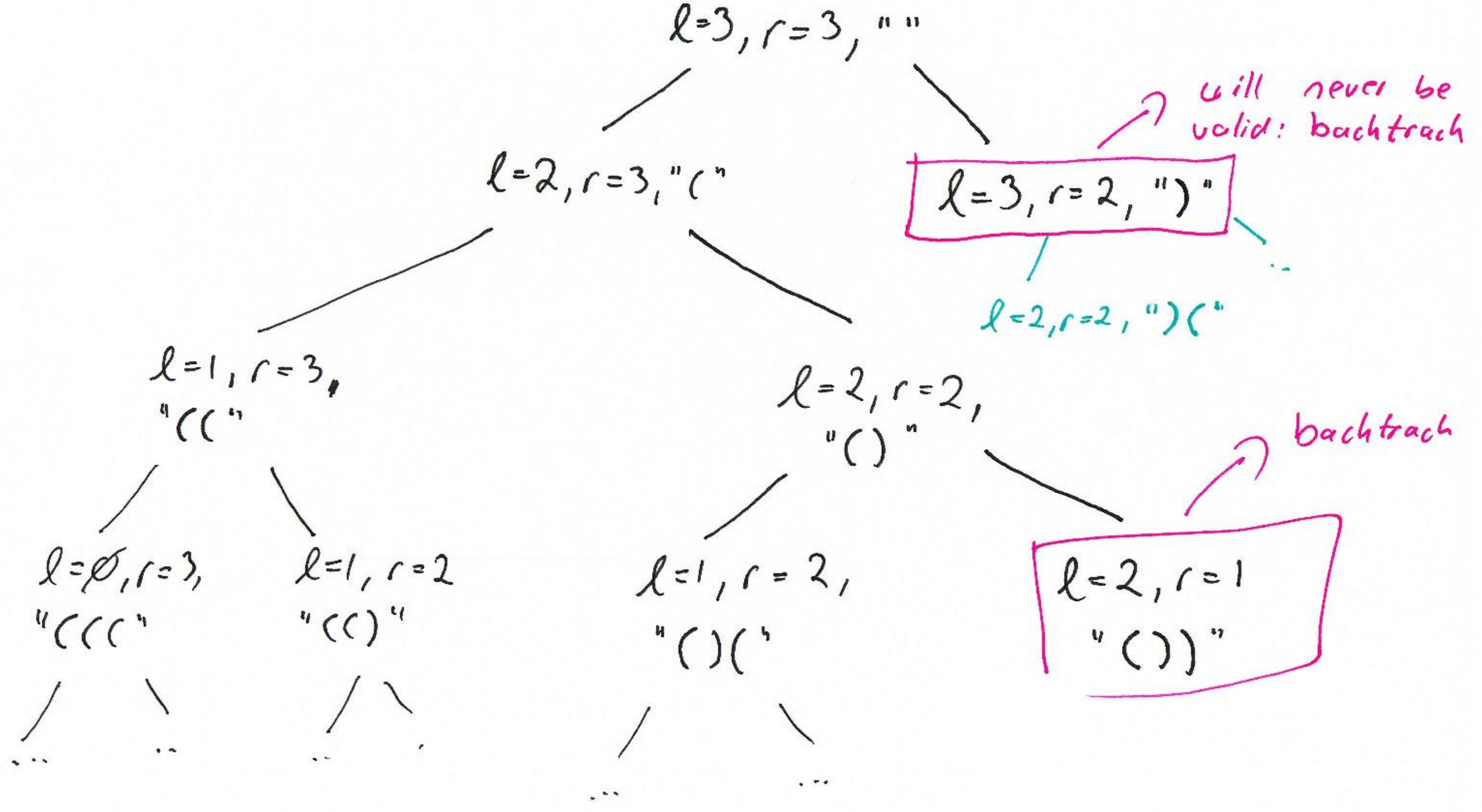
1st idea build all strings with six parantheses
"((((((", "((((()", "(((()(",… $O(2^{2n})$
in the base case, check if a pair is valid

Observation: no valid result can start with a closing parantheses: ")…"

recognizing that a state will <u>never</u> be valid and stopping early is called backtracking

16 Backtracking

Keep track of left parantheses still available, right parantheses still available



17 Well-formed parantheses

```
List<String> parantheses (int k) {
    var results = new ArrayList<String>();
    helper ("", k, k, results);
    return results;
}
void helper (String slate, int lft, int ~~rg~~ rgt, List<String> results) {
    if (rgt < lft || lft < 0 || rgt < 0) {
        return;
    }
    if (lft == 0 && rgt == 0) {
        results.add(slate);
        return;
    }
    helper(slate + "(", lft-1, rgt, results);
    helper(slate + ")", lft, rgt-1, results);
}
```

backtracking: stop before we have reached the end

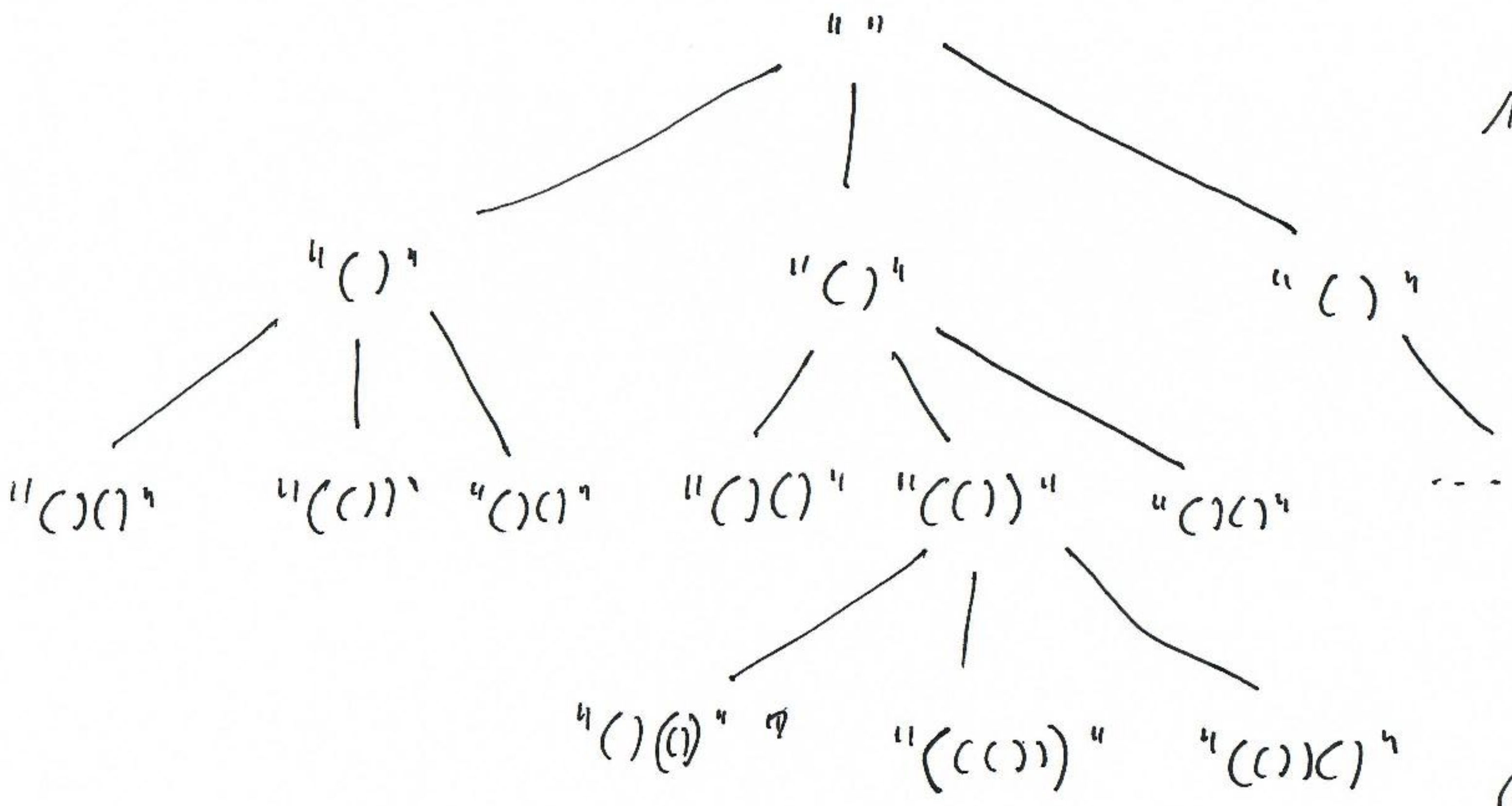
base case: we have reached the end (filled the slate)

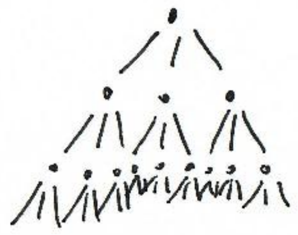
no copy since „String" is immutable

recursive calls

(18) Better bound:

Use this algorithm: for every state: put a pair in front, behind, and around





~~")))((("~~

$O(3^k)$

tigther than

$O(2^{2k})$

(19) ~~Practi~~ Phone Number Strings

ABC 1	DEF 2	GHI 3
JKL 4	MNO 5	PQR 6
STU 7	VWX 8	YZ 9
	~~4~~ Ø	

~~Given the name of a business,~~
~~reto return all valid~~ phone ~~numbers~~:

Example: ~~name = "~~

Given a phone number find all valid business names for that number.

Example: number = "91" output = ["YA", "YB
"YC", "ZA",
"ZB", "ZC"]

nuber = '3Ø1' output = [ ]

(20) Phone numbers:

```
                    MAPPING
final static char[][] = { {}, //∅
                          {'A', 'B', 'C'}, //1
                          {'D', 'E', 'F'}, ... ~~[illegible]~~, {'Y', 'Z'}};

List<String> numbers(String input){
    var results = new ArrayList<>();
    helper( ~~new~~ input, ∅ , new char[input.length()], results);
    return results;
}
void helper(~~input~~ String input, int pos, char[] slate, List<String> results){
    if(pos >= input.length()){
        results.add(new String(slate));
        return;
    }
    int digit = ~~input~~ (int)(input.charAt(pos) - '∅');
    if (digit < ∅ || digit > 9) return;
    for (char ch : MAPPING[digit]) {
        slate[pos] = ch;
        helper(input, pos+1, slate, results);
    }
}
```

(21) ~~Ex~~ Invalid Input

```
void foo( String s, List<Integer> l, int[][] a){
    validateInput( s, l, a);
    ⋮
```