Revision - Recursion

Relevel
by Unacademy

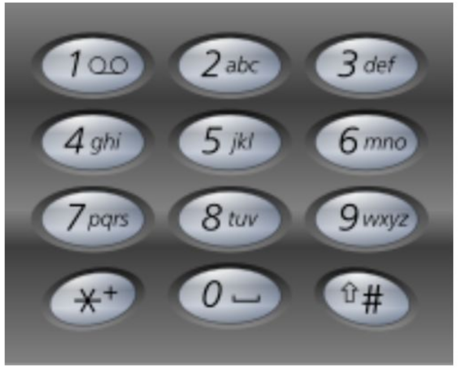
Question - 1:

Letter combinations of keypad numerics

John is very fond of cell phones. As a result of his fascination for cell phones, he gave the following question to his students.

Given a string “digits” containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. We can return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is shown below. Please note that '1' does not map to any letters.



Question - 1:

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

Example 3:

```
Input: digits = "2"
Output: ["a","b","c"]
```

Question - 1:

Algorithm :

Given a numeric string(0 not allowed), we must print all possible letter combinations. This is the cream of a recursive problem!

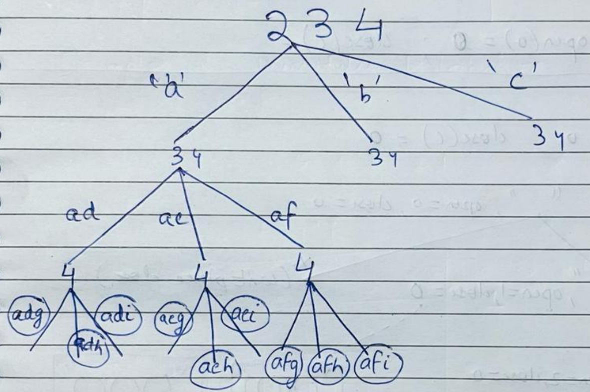
Let's take an example. Input string = "23".
Let's see the characters corresponding to "2" => ['a','b','c'].
Standing at 2, we will make 3 recursive calls.One for 'a', second for 'b', third for 'c'.
Let's analyse any one branch. Let's take 'b'.
This 'b' will be prepended to all the combinations obtained from string starting with 3.
In other words, 'b' + ['d', 'e', 'f']. This gives us 3 strings ["bd", "be", "bf"].
Similarly from 'a', we will get ["ad","ae","af"] and from 'c', we will get ["cd","ce","cf"].

Dry Run adds more clarity to the text above!

Question - 1:

Input String = 234

2 → 'a', 'b', 'c'
3 → 'd', 'e', 'f'
4 → 'g', 'h', 'i'



Similiarly we will get 9 combinations from 'b' branch and 9 combinations from 'c' branch.

Question - 1:

Time Complexity:

Let’s take input size as n.

‘7’ has maximum characters mapped to itself -> [‘w’,’x’,’y’,’z’]. Size of this is 4.

Worst case input - 777….

Total computations - 4*4*..n times

Time Complexity - O(4^n)

Space Complexity:

Considering recursion call stack, space complexity => O(n) where n is the length of the input passed.

Question - 2:

Generate all possible, valid parentheses

Adam is a parentheses freak! He is an expert in solving parentheses problems. As an exercise, his father gave him the following problem.

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

Input : n = 3

Output : [“((()))”, “(()())”, “()(())”, “(())()”, “()()()”]

Example 2:

Input : n = 1

Output : [“()”]

Question - 2:

Algorithm :

1. We have to form n pairs of parentheses.
2. We are going to approach this recursively. The agenda is to arrange n open brackets, and n closed brackets to balance the final expression.
3. Initially, we had 0 open and 0 closed. Let us start our algorithm.

4. At every step, we have two options -
    a. We put an open bracket.
    b. We put a closed bracket.
5. Questions arising - When can we put an open bracket? When can we put a closed bracket?

Question - 2:

6. Three possible cases -
   a. open > close
   b. open = close
   c. open < close
7. As discussed, we have to furnish balanced bracket expressions. So with that logic, 6 c) would never be a possibility.
8. So we can have two possibilities
   a. open > close
   b. open = close

Considering a., when open > close, we have two options for our current slot.

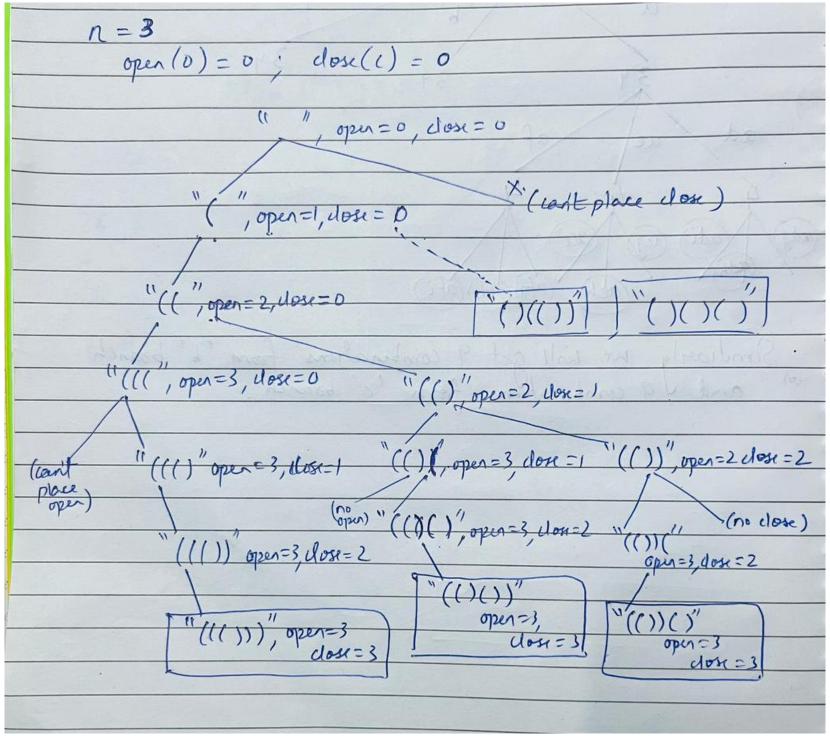
We can either place an open bracket or place a closed bracket.

Considering b., when open = close, we have only one option: to place an open bracket. If we place a closed bracket, the expression will not be valid.

Question - 2:

9. Making observations from 7. more concise, we can say that -
    a. we can place open when -
        A. open < n
        B. open > close
    b. we can place close when -
        A. open > close
10. Bingo! we are done.

Question - 2:



n = 3
open (0) = 0 ; close (0) = 0
" ", open = 0, close = 0
X (can't place close)
"(", open=1, close = 0
"((", open=2, close=0
"()(())"
"()()()"
"(((", open=3, close=0
"(()", open=2, close=1
(can't place open)
"((()" open=3, close=1
"(()(", open=3, close =1
"(())", open=2 close =2
(no open)
"(()()", open=3, close=2
(no close)
"((())" open=3, close= 2
"(())(" open=3, close=2
"((()))", open=3 close=3
"(()())" open=3, close=3
"(())()" open=3 close=3

Question - 2:

Time Complexity: O(number of arrangements possible)
For interested readers - The upper bound is equal to the nth Catalan number.

Space Complexity:
Considering recursion call stack, space complexity => O(n) where 2n is the length of the parenthesis expression.

Question - 3:

Decode numeric string

Given a string containing only digits, return the number of ways to decode it. We are essentially trying to obtain the decodings using the following mapping.

1 => “A”, 2 => “B”, ……….Z => “26”

Examples

Input : s = “12”

Output : 2

Explanation : 12 can be decoded as “AB” and “L”.

Input : s = “226”

Output : 3

Explanation : “226” can be decoded as “BBF”, “VF”, “BZ”.

Note :

Input : s = “0”, “06”, “004” are all invalid inputs as no character has an integer decoding with starts with 0.

Question - 3:

Algorithm :

As usual, we are going to be lazy and will rely upon the immense power of recursion to pull this off!

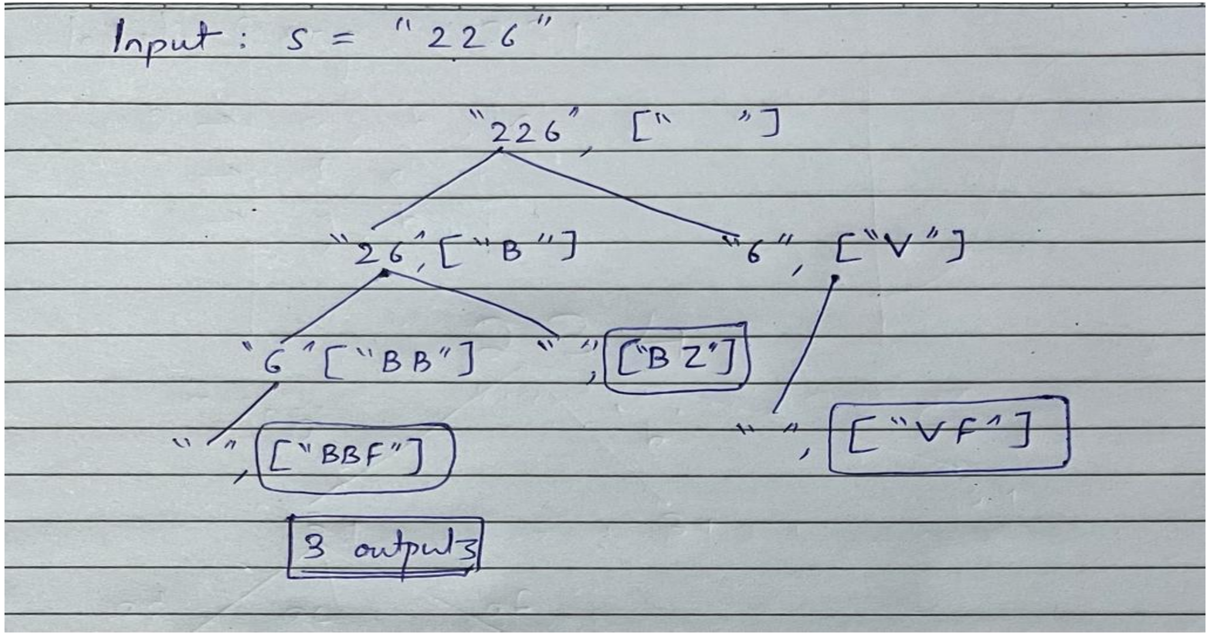
At every index, we have 2 options -

A. Consume the current number(if it is non-zero and belongs to [1,9]).
B. Consume the current and the next(if the 2 digit number belongs to [10,26]).

Both these options can be added to give the final answer.

Question - 3:

Dry Run for s=”226”



Question - 3:

Time Complexity:

Consider the input as “1111…”. At step i, there would be 2^i calls.

1 + 2 + 4 + 8 + …..2^n => This is a GP.

Adding it all, we get an expression where the highest power is 2^n.

Hence time complexity - O(2^n).

Space Complexity:

Considering recursion call stack, space complexity => O(n) where n is the length of the input passed.

Question - 4:

String Pattern Matching

Suzie loves solving pattern-matching puzzles. She is an expert!

In order to test her competence, her father asks her to solve the following question.

Given an input string (str) and a pattern (ptr), implement pattern matching with support for '?' and '*' where:

'?' Matches any single character.

'*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

Examples

Input : str = “ab”, ptr = “b”
Output : False
Explanation : pattern “a” doesn’t match the entire string str.

Input : str = “bbb”, ptr = “*”
Output : True
Explanation : pattern “*” matches any sequence of characters.

Question - 4:

We have to validate whether the input string can be consumed by the pattern string entirely or not.
The brute force would be a janky one here. If anything, we can try to obtain all possible configurations of the ptr string. If any of those match with str string, we can say that the answer is true.
However, that is going to be cumbersome.

Algorithm:
Alright, so we have different scenarios to address. Let's look at them one by one.
str -> input string ; ptr -> pattern string ; si -> pointer over str, pi -> pointer over ptr

1. if (str[si] == ptr[pi]), then we can move ahead by 1 in both strings.
2. if(ptr[pi] == '?'), then we can permit this '?' to cover whatever is the value of str[si]. Hence we can move ahead by 1 in both strings.

Question - 4:

3. Now coming to the most fancy case, if(ptr[pi] == ‘*’). Here this ‘*’ can play two roles.
    a. ptr[pi] is considered as a match for an empty string. In this case, we do not move ahead in string str, but we move ahead in string ptr. This is because ‘*’ has been used for empty substring.
    b. ptr[pi] is considered a match for str[si]. In this case, we are essentially matching ‘*’ to a sequence of characters. So we will move ahead in string str but not in string ptr. As this ‘*’ can be used for consuming more characters ahead of str[si].

Our answer would be true if either of a or b is true.

Question - 4:

Time Complexity:

If ptr = “*****...”, and str is any valid string, then at each step there would be 2^step function calls, Again this would be a GP. The height of the recursion tree would be equal to max(ptr.length, str.length) Hence TC => O(2^n).

Space Complexity:

Considering recursion call stack, space complexity => O(max(n,m)), where n is the length of str and m is the length of ptr.

Question - 5:

Upper-Lower Permutations

Given a string str, one can convert every letter(not numeric, obvio!) into lowercase or uppercase and create another string.

Return a list of all possible strings that could be created. Return the output in any order.

Examples

Input: s = "a1b2"

Output: ["a1b2","a1B2","A1b2","A1B2"]

Input: s = "3z4"

Output: ["3z4","3Z4"]

Question - 5:

Another question where we have to explore all possible scenarios exhaustively. What is the laziest yet the most rewarding(less input, more output) method of exploring all options? You got it right!
It’s Recursion!

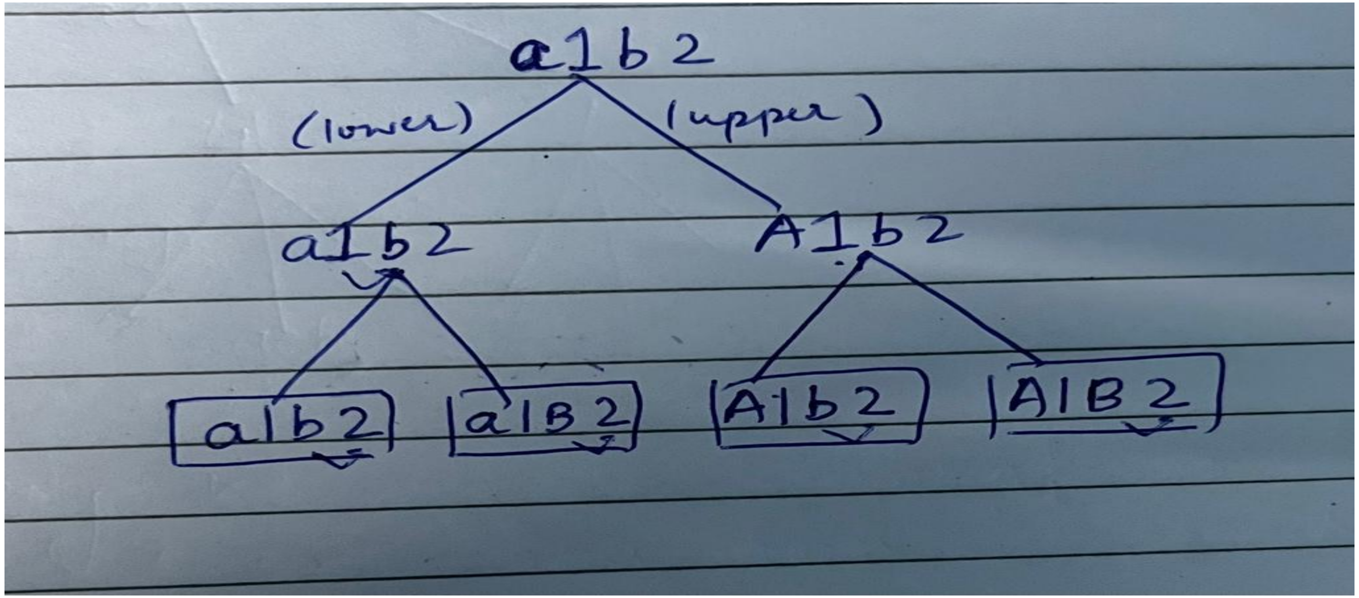
Algorithm:
s[i] can either be a digit or an alphabet. AS we have nothing to worry about; we can move ahead.
What if s[i] is an alphabet? Then we have to deal with 2 possible scenarios.

1. The first call - move ahead by using s[i] in lowercase.
2. The second call - move ahead by using s[i] in uppercase.

These two calls will follow their recursive course and contribute to the final list of arrangements at the end(which string is over).

Question - 5:

Here's a dry for s = "a1b2".



Question - 5:

Time Complexity:

Consider a string containing all alphabets. Each node will spawn two children. We will have a recursion tree where level i has 2^i nodes.

TC for such a scenario ->O(2^n), where is the length of the input string.

Space Complexity:

Considering recursion call stack, space complexity => O(n) where n is the length of the input passed.

Practise Questions:

1. **Match the REGEX**

Let’s build a regex matcher. Given an input string s and a pattern p, implement regular expression matching with support for '.' and '*' where:

'.' Matches any single character.
'*' Matches zero or more of the preceding element.
The matching should cover the entire input string (not partial).

Practise Questions:

Examples:

Input: s = "aa", p = "a*"

Output: true

Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Input: s = "ab", p = ".*"

Output: true

Explanation: ".*" means "zero or more (*) of any character (.)".

Practise Questions:

2. **Put the right Commas!**

Marie is always in a hurry. She wrote down many positive integers in a string called str. However, she realized that she forgot to add commas to separate the different numbers. She remember that the list of integers was non-decreasing and that no integer had leading zeros. You have to help Marie.

Return the number of possible lists of integers that she could have written down to get the string str. Since the answer may be large, return its modulo with 1e9 + 7.

Practise Questions:

Examples:

Input: num = "327"

Output: 2

Explanation: Marie could have written down the numbers:

3, 27

327

Input: num = "094"

Output: 0

Explanation: No numbers can have leading zeros and all numbers must be positive.

Thank you!