320. Generalized Abbreviation

Dec. 9, 2016 | 11.8K views

Write a function to generate the generalized abbreviations of a word.

Note: The order of the output does not matter.

Example:

```
Input: "word"
Output:
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1",
```

Summary

This article is for intermediate readers. It introduces the following ideas: Backtracking and Bit Manipulation

Solution

Approach #1 (Backtracking) [Accepted]

Intuition

How many abbreviations are there for a word of length n? The answer is 2^n because each character can either be abbreviated or not, resulting in different abbreviations.

Algorithm

The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in several choices to give all the possible solutions to the problem. The completion is done incrementally, by extending the candidate in many steps. Abstractly, the partial candidates can be seen as nodes of a tree, the potential search tree. Each partial candidate is the parent of the candidates that derives from it by an extension step; the leaves of the tree are the partial candidates that cannot be extended any further.

In our problem, the partial candidates are incomplete abbreviations that can be extended by one of the two choices:

1. keep the next character;
2. abbreviate the next character.

We extend the potential candidate in a depth-first manner. We backtrack when we reach a leaf node in the search tree. All the leaves in the search tree are valid abbreviations and shall be put into a shared list which will be returned at the end.

Java

Copy

```
public class Solution {
    public List<String> generateAbbreviations(String word){
        List<String> ans = new ArrayList<String>();
        backtrack(ans, new StringBuilder(), word, 0, 0);
        return ans;
    }

    // i is the current position
    // k is the count of consecutive abbreviated characters
    private void backtrack(List<String> ans, StringBuilder builder, String word, int i, int k){
        int len = builder.length(); // keep the length of builder
        if(i == word.length()){
            if (k != 0) builder.append(k); // append the last k if non zero
            ans.add(builder.toString());
        } else {
            // the branch that word.charAt(i) is abbreviated
            backtrack(ans, builder, word, i + 1, k + 1);

            // the branch that word.charAt(i) is kept
            if (k != 0) builder.append(k);
            builder.append(word.charAt(i));
            backtrack(ans, builder, word, i + 1, 0);
        }
        builder.setLength(len); // reset builder to the original state
    }
}
```

Complexity Analysis

- Time complexity : $O(n2^n)$. For each call to `backtrack`, it either returns without branching, or it branches into two recursive calls. All these recursive calls form a complete binary recursion tree with 2^n leaves and $2^n - 1$ inner nodes. For each leaf node, it needs $O(n)$ time for converting builder to String; for each internal node, it needs only constant time. Thus, the total time complexity is dominated by the leaves. In total that is $O(n2^n)$.
- Space complexity : $O(n)$. If the return list doesn't count, we only need $O(n)$ auxiliary space to store the characters in `StringBuilder` and the $O(n)$ space used by system stack. In a recursive program, the space of system stack is linear to the maximum recursion depth which is n in our problem.

Approach #2 (Bit Manipulation) [Accepted]

Intuition

If we use 0 to represent a character that is not abbreviated and 1 to represent one that is. Then each abbreviation is mapped to an n bit binary number and vice versa.

Algorithm

To generate all the 2^n abbreviation with non-repetition and non-omission, we need to follow rules. In approach #1, the rules are coded in the backtracking process. Here we introduce another way.

From the intuition section, each abbreviation has a one to one relationship to a n bit binary number x. We can use these numbers as blueprints to build the corresponding abbreviations.

For example:

Given word = `"word"` and x = `0b0011`

Which means `'w'` and `'o'` are kept, `'r'` and `'d'` are abbreviated. Therefore, the result is "wo2".

Thus, for a number x, we just need to scan it bit by bit as if it is an array so that we know which character should be kept and which should be abbreviated.

To scan a number x bit by bit, one could extract its last bit by `b = x & 1` and shift x one bit to the right, i.e. `x >>= 1`. Doing this repeatedly, one will get all the n bits of x from last bit to first bit.

Java

Copy

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> ans = new ArrayList<>();
        for (int x = 0; x < (1 << word.length()); ++x) // loop through all possible x
            ans.add(abbr(word, x));
        return ans;
    }

    // build the abbreviation for word from number x
    private String abbr(String word, int x) {
        StringBuilder builder = new StringBuilder();
        int k = 0, n = word.length(); // k is the count of consecutive ones in x
        for (int i = 0; i < n; ++i, x >>= 1) {
            if ((x & 1) == 0) { // bit is zero, we keep word.charAt(i)
                if (k != 0) { // we have abbreviated k characters
                    builder.append(k);
                    k = 0; // reset the counter k
                }
                builder.append(word.charAt(i));
            }
            else // bit is one, increase k
                ++k;
        }
        if (k != 0) builder.append(k); //don't forget to append the last k if non zero
        return builder.toString();
    }
}
```

Complexity Analysis

- Time complexity : $O(n2^n)$. Building one abbreviation from the number x, we need scan all the n bits. Besides the `StringBuilder::toString` function is also linear. Thus, to generate all the 2^n, it costs $O(n2^n)$ time.
- Space complexity : $O(n)$. If the return list doesn't count, we only need $O(n)$ auxiliary space to store the characters in `StringBuilder`.

Rate this article: