320. Generalized Abbreviation Medium **Bit Manipulation** String **Backtracking**

Problem Description

The problem asks us to generate all possible generalized abbreviations for a given word. A generalized abbreviation for a word is a transformation where any number of non-overlapping and non-adjacent substrings can be replaced with their respective lengths. The key points to remember are:

- Substrings that we replace must not be adjacent.

Substrings that we replace with numbers must not overlap with each other.

 We can use the whole word as a substring, which means the entire word can be replaced with a single number indicating its length. The returned list does not need to be in any specific order.

To understand it better, let's look at an example with the word "abc". Possible generalized abbreviations for "abc" include "abc" (no

change), "1bc" (replacing "a" with "1"), "a2" (replacing "bc" with "2"), "ab1" (replacing "c" with "1"), "1b1" (replacing the first and last characters with "1"), and "3" (replacing the entire word with "3").

Intuition

The process to solve this problem can be approached using recursion and backtracking. We can think of each character in the word

as a decision point: at each character, we can decide to: Replace a substring starting at this character with a number that represents its length. Skip the character to keep it as is.

While performing these operations, we must ensure that we are not violating the rules of replacing non-overlapping and non-

- adjacent substrings. Therefore, each time we replace a substring with its length, we skip the next character to maintain the nonadjacency condition.
- To arrive at the solution, we start by writing a recursive function that can generate the aforementioned abbreviations by considering

the remaining string after this character.

each character. This function should have the ability to record the current state (as an abbreviation in progress) and be able to backtrack to explore different abbreviation possibilities until all possibilities have been exhausted. The algorithm works as follows:

1. At each recursion level, the function is given the remaining string to process and the current abbreviation state. 2. If the remaining string is empty, the current state is one of the solutions, so add it to the answer list. 3. For each possible length of the substring to be abbreviated (from 1 to the length of the remaining string): Append the length number to the current state. If there are characters remaining after the substring, append the next character to enforce non-adjacency, and recurse with

- Backtrack: remove the added elements from the current state to explore other abbreviation possibilities. 4. Also, consider keeping the current character as is and recurse with the remaining string after this character.
- The solution to the problem emerges as we systematically explore each abbreviation possibility while adhering to the established
- rules. Solution Approach

The implemented solution uses depth-first search (DFS) and backtracking to generate all possible generalized abbreviations of the

input word. The code defines a recursive function dfs that accepts the remaining string s to process and a temporary list t which keeps track of the current abbreviation being constructed.

Here is a step-by-step walkthrough of the implementation: 1. Base Case: If the remaining string s is empty, it means we have reached the end of the string and have constructed a valid

abbreviation. Hence, we join the elements in the temporary list t into a string and append it to ans, the final list of abbreviations.

recursion.

1 def dfs(s, t):

13

14

15

return

t.append(s[0])

dfs(s[1:], t)

2. Recursive Case:

replacement. For each length i, we have two cases to consider: ■ Abbreviation: We append the length i to t as a substring replacement. If i is less than the length of s, we also append the next character to t (to satisfy the non-adjacency condition) and then recursively call dfs with the remaining string

■ No Abbreviation (End of Recursion): If the length i is equal to the length of s, we simply perform a recursive call

without adding any more characters, as we would be considering the entire remaining string as one substring.

s[i+1:]. After the recursive call, we backtrack by popping the added elements from t.

• We iterate over the range from 1 to len(s) + 1 which represents the length of the substring we are considering for

 Additionally, we must consider the case where we do not replace the current character with its abbreviation. Therefore, we append the current character s[0] to t and recursively call dfs with s[1:]. We backtrack by popping the character after

The choice of algorithm and data structure:

explore other paths in the decision tree.

ans.append(''.join(t))

for i in range(1, len(s) + 1):

t.pop() # Backtrack

t.append(str(i))

if i < len(s):</pre>

space. • Final List (ans): Used to store all the possible abbreviation combinations generated. Here is the code excerpt that illustrates the recursive approach:

• Temporary List (t): It keeps track of the current state of the abbreviation and is modified in place during recursion to save

• Depth-First Search (DFS): Used to explore all possible abbreviation combinations by diving deeper into the recursive tree

• Backtracking: Used to undo the decision at each step (once all possibilities after that decision are explored) to backtrack and

- t.append(s[i]) dfs(s[i + 1:], t)t.pop() # Backtrack else: dfs(s[i:], t) # End of Recursion
- t.pop() # Backtrack 16

Let's assume our given word is "word". We use the recursive implementation of the solution approach to generate all possible

The function generateAbbreviations initializes the final answer ans and calls dfs with the initial word and an empty temporary list to

Example Walkthrough

start the process. After all the possible abbreviations are generated, the ans list is returned.

generalized abbreviations. We'll use a step-by-step walk through to illustrate how the algorithm works.

structure based on the decisions made (to replace or not to replace substrings).

```
First, we call the dfs function with the full string "word" and an empty temporary list t.
  1. In the first level of recursion, s is "word".

    We consider all lengths of substrings starting from 1 to the length of "word".

    a. Length 1: We replace "w" with "1". - Temp list t becomes ["1"]. - We call dfs with the remaining string "ord" and t as ["1", "o"]
    (enforcing non-adjacency). - This recursion continues, similarly replacing one character at a time and enforcing non-adjacency,
    until the base case is reached and the possible abbreviations like "1o2", "1or1", and "1ord" are added to ans.
    b. Length 2: We replace "wo" with "2". - Temp list t becomes ["2"]. - We call dfs with the remaining string "rd" and t as ["2", "r"]
    (enforcing non-adjacency). - Following the similar pattern, "2r1" and "2rd" are added to ans.
```

"w1r1", "w2", "wo1d", "wor1", and "word".

adjacent, maintaining the problem's constraints.

if not remaining:

return

else:

return answers

private List<String> abbreviations;

abbreviations = new ArrayList<>();

dfs(word, currentAbbreviation);

if ("".equals(remainingSubstring)) {

return abbreviations;

return;

} else {

Java Solution

15

16

17

18

19

20

21

22

34

35

36

37

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

39

40

42

43

44

45

46

47

48

49

50

51

52

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

62

63

64

65

67

68

69

70

71

72

73

74

76

8

9

10

11

75 };

public:

string result;

return result;

for (const auto& part : parts) {

result += part;

abbreviations.clear();

return abbreviations;

result += delimiter;

vector<string> currentAbbreviation;

// Global variable to store the final list of abbreviations.

function generateAbbreviations(word: string): string[] {

let currentAbbreviation: string[] = [];

// Function that initiates the generation of abbreviations for a word.

// Temporary array to keep track of current abbreviation progression.

// Start the depth-first search (DFS) from the beginning of the word.

dfs(word, currentAbbreviation);

Temp list t is ["w"].

"3d", "w3", and "4", among others.

2. Additionally, we have the option to keep "w" as is and recurse with the remaining string "ord".

These steps will systematically continue for each possibility, exploring further branches in the recursion tree and using backtracking

generalized abbreviations. The recursive calls ensure that we don't create overlapping abbreviations and that no two numbers are

The final output for the given word "word" will hence include entries like "word", "10rd", "w1rd", "w01d", "wor1", "2rd", "w2d", "w02",

to generate all possible abbreviations. Once all recursion paths are explored, we have our complete list ans with all generated

This recursion follows the same pattern as before, exploring all options with the prefix "w" and generating abbreviations like

Python Solution class Solution:

answers.append(''.join(current_abbreviation))

current_abbreviation.append(remaining[i])

Backtrack to previous state

answers = [] # List to store all possible abbreviations

// Public method that initiates the generation of abbreviations for a word.

// Temporary list to keep track of current abbreviation progression.

// Start the depth first search (DFS) from the beginning of the word.

// Return the list of all possible abbreviations after DFS is complete.

// Helper method to perform depth first search to generate all abbreviations.

abbreviations.add(String.join("", currentAbbreviation));

for (int i = 1; i <= remainingSubstring.length(); ++i) {</pre>

// Add number abbreviation for the current length.

private void dfs(String remainingSubstring, List<String> currentAbbreviation) {

// If the remaining substring is empty, join all collected parts and add to result.

currentAbbreviation.add(String.valueOf(remainingSubstring.charAt(i)));

// If no characters are left, continue DFS with the current abbreviation.

// Try keeping the first character as is and continue DFS for the rest of the string.

// Backtracking - remove the number abbreviation to try a different abbreviation length.

dfs(remainingSubstring.substring(i + 1), currentAbbreviation);

currentAbbreviation.remove(currentAbbreviation.size() - 1);

dfs(remainingSubstring.substring(i), currentAbbreviation);

currentAbbreviation.remove(currentAbbreviation.size() - 1);

// Backtracking - remove the last character to explore new paths.

// Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).

public List<String> generateAbbreviations(String word) {

List<String> currentAbbreviation = new ArrayList<>();

current_abbreviation.pop()

dfs(remaining[i + 1:], current_abbreviation)

def generateAbbreviations(self, word: str) -> List[str]:

def dfs(remaining, current_abbreviation):

if i < len(remaining):</pre>

c. This pattern continues for length 3 and 4, adding "3d" and "4" respectively to ans.

10 # Consider abbreviation for every possible length starting from each index 11 for i in range(1, len(remaining) + 1): # Append the number (length of abbreviation) 12 13 current_abbreviation.append(str(i)) 14

Helper function to perform depth-first search to find all possible abbreviations

No more characters left to abbreviate, add the current abbreviation to answers

If there are characters remaining, append the next character after abbreviation

All characters are abbreviated, call dfs with an empty remaining string

dfs(word, []) # Start the depth-first search with the full word and empty current abbreviation

- 23 dfs(remaining[i:], current_abbreviation) 24 25 # Backtrack to consider next abbreviation possibility 26 current_abbreviation.pop() 27 28 # Consider not abbreviating the first character and recursively find further abbreviations 29 current_abbreviation.append(remaining[0]) dfs(remaining[1:], current_abbreviation) 30 # Backtrack after exploring abbreviations with the first character included 31 32 current_abbreviation.pop() 33
- 1 import java.util.ArrayList; import java.util.List; class Solution { // Class variable to store the final list of abbreviations.

33 currentAbbreviation.add(Integer.toString(i)); 34 // If there are characters left, add the next character and continue DFS. 35 36 if (i < remainingSubstring.length()) {</pre> 37 38

```
currentAbbreviation.add(String.valueOf(remainingSubstring.charAt(0)));
53
           dfs(remainingSubstring.substring(1), currentAbbreviation);
54
55
           // Backtracking - remove the character to keep the path tree clean for new paths.
           currentAbbreviation.remove(currentAbbreviation.size() - 1);
56
57
58 }
59
C++ Solution
  1 #include <vector>
  2 #include <string>
     using namespace std;
    class Solution {
     private:
         vector<string> abbreviations; // Vector to store the final list of abbreviations.
  9
         // Helper method to perform depth-first search to generate all abbreviations.
 10
         void dfs(const string& remainingSubstring, vector<string>& currentAbbreviation) {
 11
 12
             // If the remaining substring is empty, concatenate all collected parts and add to the result.
 13
             if (remainingSubstring.empty()) {
 14
                 abbreviations.push_back(join(currentAbbreviation, ""));
 15
                 return;
 16
 17
             // Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).
 18
 19
             for (int i = 1; i <= remainingSubstring.size(); ++i) {</pre>
 20
                 // Add number abbreviation for the current length.
 21
                 currentAbbreviation.push_back(to_string(i));
 22
 23
                 // If there are characters left, add the next character and continue DFS.
                 if (i < remainingSubstring.size()) {</pre>
 24
 25
                     currentAbbreviation.push_back(string(1, remainingSubstring[i]));
 26
                     dfs(remainingSubstring.substr(i + 1), currentAbbreviation);
 27
                     // Backtracking - remove the last character to explore new paths.
 28
 29
                     currentAbbreviation.pop_back();
                 } else {
 30
 31
                     // If no characters are left, continue DFS with the current abbreviation.
 32
                     dfs(remainingSubstring.substr(i), currentAbbreviation);
 33
 34
 35
                 // Backtracking - remove the number abbreviation to try a different abbreviation length.
 36
                 currentAbbreviation.pop_back();
 37
 38
 39
             // Try keeping the first character as is and continue DFS for the rest of the string.
             currentAbbreviation.push_back(string(1, remainingSubstring[0]));
 40
             dfs(remainingSubstring.substr(1), currentAbbreviation);
 41
 42
 43
             // Backtracking — remove the character to keep the path tree clean for new paths.
 44
             currentAbbreviation.pop_back();
 45
```

// Helper method to join strings in a vector into a single string with a given delimiter.

if (&part != &parts[0]) { // Avoid adding delimiter before the first element.

string join(const vector<string>& parts, const string& delimiter) {

// Iterating over each part and appending to the result string.

// Public method that initiates the generation of abbreviations for a word.

// Temporary vector to keep track of the current abbreviation progression.

// Return the vector of all possible abbreviations after DFS is complete.

// Start the depth-first search (DFS) from the beginning of the word.

vector<string> generateAbbreviations(const string& word) {

dfs(word, currentAbbreviation); 12 13 14 // Return the list of all possible abbreviations after DFS is complete. 15 return abbreviations; 16 } 17

Typescript Solution

let abbreviations: string[] = [];

abbreviations = [];

```
// Helper function to perform depth-first search to generate all abbreviations.
   function dfs(remainingSubstring: string, currentAbbreviation: string[]): void {
       // If the remaining substring is empty, join all collected parts and add to the result.
20
       if (remainingSubstring === "") {
           abbreviations.push(currentAbbreviation.join(""));
           return;
24
25
26
       // Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).
       for (let i = 1; i <= remainingSubstring.length; i++) {</pre>
           // Add number abbreviation for the current length.
28
           currentAbbreviation.push(i.toString());
29
           // If there are characters left, add the next character and continue DFS.
32
           if (i < remainingSubstring.length) {</pre>
               currentAbbreviation.push(remainingSubstring.charAt(i));
33
               dfs(remainingSubstring.substring(i + 1), currentAbbreviation);
34
35
36
               // Backtracking — remove the last character to explore new paths.
               currentAbbreviation.pop();
37
           } else {
38
               // If no characters are left, continue DFS with the current abbreviation.
39
               dfs("", currentAbbreviation);
40
           // Backtracking - remove the number abbreviation to try a different abbreviation length.
           currentAbbreviation.pop();
44
45
46
       // Try keeping the first character as is and continue DFS for the rest of the string.
47
       currentAbbreviation.push(remainingSubstring.charAt(0));
48
       dfs(remainingSubstring.substring(1), currentAbbreviation);
49
50
       // Backtracking - remove the character to keep the path tree clean for new paths.
51
52
       currentAbbreviation.pop();
53 }
54
Time and Space Complexity
The time complexity of this code can be analyzed by considering the number of different abbreviation combinations that can be
```

generated for a word of length n. At every character position in the word, we have two choices: abbreviate by using a number or use the character itself. This results in a recursion tree with a maximum branch factor of n, as we can abbreviate up to n characters in one go, forming a sequence of overlapping subproblems. Therefore, the time complexity is 0(2^n) since the recursion depth is n and

at every level we are making two choices. However, because of the way the abbreviations are generated (with numbers that can represent multiple characters), the actual time complexity is less strictly bounded and is dependent on the structure of the recursion. The space complexity is determined by the depth of the recursion stack and the space required to store the intermediate strings. In the worst case, the recursion depth is n and the intermediate strings can also be of length n, leading to a space complexity of O(n).