Unique Generalized Abbreviations (hard)

We'll cover the following

Problem Statement

Given a word, write a function to generate all of its unique generalized abbreviations.

A generalized abbreviation of a word can be generated by replacing each substring of the word with the count of characters in the substring. Take the example of "ab" which has four substrings: "", "a", "b", and "ab". After replacing these substrings in the actual word by the count of characters, we get all the generalized abbreviations: "ab", "1b", "a1", and "2".

Note: All contiguous characters should be considered one substring, e.g., we can't take "a" and "b" as substrings to get "11"; since "a" and "b" are contiguous, we should consider them together as one substring to get an abbreviation "2".

Example 1:

```
Input: "BAT"
Output: "BAT", "BA1", "B1T", "B2", "1AT", "1A1", "2T", "3"
```

Example 2:

```
Input: "code"
Output: "code", "cod1", "co1e", "co2", "c1de", "c1d1", "c2e", "c3", "1ode", "1od1", "1o1e", "1o2",
"2de", "2d1", "3e", "4"
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```python
def generate_generalized_abbreviation(word):
  result = []
  # TODO: Write your code here
  return result


def main():
  print("Generalized abbreviation are: " +
        str(generate_generalized_abbreviation("BAT")))
  print("Generalized abbreviation are: " +
        str(generate_generalized_abbreviation("code")))


main()
```

Run | Save | Reset

Solution

This problem follows the Subsets pattern and can be mapped to Balanced Parentheses. We can follow a similar BFS approach.

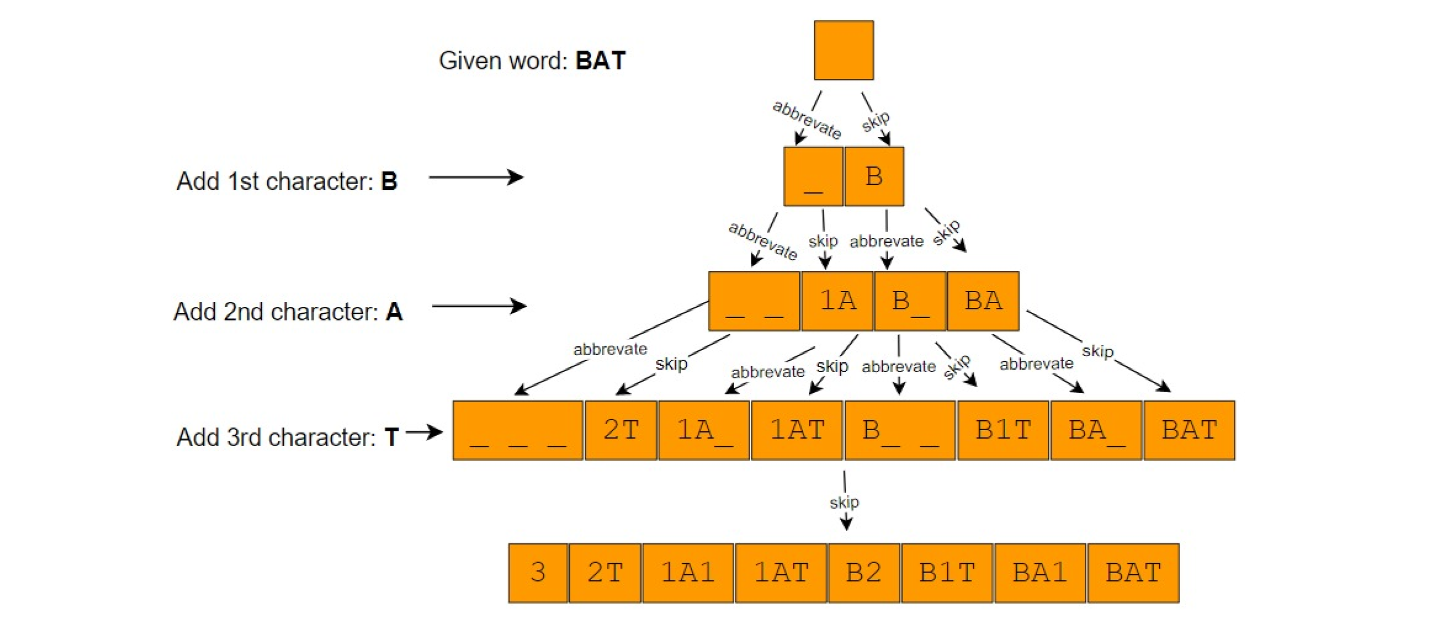
Let's take Example-1 mentioned above to generate all unique generalized abbreviations. Following a BFS approach, we will abbreviate one character at a time. At each step, we have two options:

- Abbreviate the current character, or
- Add the current character to the output and skip the abbreviation.

Following these two rules, let's abbreviate `BAT`:

1. Start with an empty word: ""
2. At every step, we will take all the combinations from the previous step and apply the two abbreviation rules to the next character.
3. Take the empty word from the previous step and add the first character to it. We can either abbreviate the character or add it (by skipping abbreviation). This gives us two new words: `_`, `B`.
4. In the next iteration, let's add the second character. Applying the two rules on `_` will give us `_ _` and `1A`. Applying the above rules to the other combination `B` gives us `B_` and `BA`.
5. The next iteration will give us: `_ _ _`, `2T`, `1A_`, `1AT`, `B _ _`, `B1T`, `BA_`, `BAT`
6. The final iteration will give us: `3`, `2T`, `1A1`, `1AT`, `B2`, `B1T`, `BA1`, `BAT`

Here is the visual representation of this algorithm:



Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```python
from collections import deque


class AbbreviatedWord:

  def __init__(self, str, start,  count):
    self.str = str
    self.start = start
    self.count = count


def generate_generalized_abbreviation(word):
  wordLen = len(word)
  result = []
  queue = deque()
  queue.append(AbbreviatedWord(list(), 0, 0))
  while queue:
    abWord = queue.popleft()
    if abWord.start == wordLen:
      if abWord.count != 0:
        abWord.str.append(str(abWord.count))
      result.append(''.join(abWord.str))
    else:
      # continue abbreviating by incrementing the current abbreviation count
      queue.append(AbbreviatedWord(list(abWord.str),
                                   abWord.start + 1, abWord.count + 1))

      # restart abbreviating, append the count and the current character to the string
      if abWord.count != 0:
        abWord.str.append(str(abWord.count))
```

Run | Save | Reset

Time complexity

Since we had two options for each character, we will have a maximum of 2^N combinations. If you see the visual representation of Example-1 closely, you will realize that it is equivalent to a binary tree, where each node has two children. This means that we will have 2^N leaf nodes and $2^N - 1$ intermediate nodes, so the total number of elements pushed to the queue will be $2^N + 2^N - 1$, which is asymptotically equivalent to $O(2^N)$. While processing each element, we do need to concatenate the current string with a character. This operation will take $O(N)$, so the overall time complexity of our algorithm will be $O(N * 2^N)$.

Space complexity

All the additional space used by our algorithm is for the output list. Since we can't have more than $O(2^N)$ combinations, the space complexity of our algorithm is $O(N * 2^N)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

Java | Python3 | C++ | JS

```python
def generate_generalized_abbreviation(word):
  result = []
  generate_abbreviation_recursive(word, list(), 0, 0, result)
  return result


def generate_abbreviation_recursive(word, abWord, start, count, result):

  if start == len(word):
    if count != 0:
      abWord.append(str(count))
    result.append(''.join(abWord))
  else:
    # continue abbreviating by incrementing the current abbreviation count
    generate_abbreviation_recursive(
      word, list(abWord), start + 1, count + 1, result)

    # restart abbreviating, append the count and the current character to the string
    if count != 0:
      abWord.append(str(count))
    newWord = list(abWord)
    newWord.append(word[start])
    generate_abbreviation_recursive(word, newWord, start + 1, 0, result)


def main():
  print("Generalized abbreviation are: " +
        str(generate_generalized_abbreviation("BAT")))
  print("Generalized abbreviation are: " +
        str(generate_generalized_abbreviation("code")))
```

Run | Save | Reset