2767. Partition String Into Minimum Beautiful Substrings String ] **Dynamic Programming** Medium Hash Table Backtracking

# **Problem Description**

conditions: 1. It does not start with a zero.

The goal is to split a given binary string s into the minimum number of contiguous substrings where each substring satisfies two

**Leetcode Link** 

If this partition isn't possible according to the above rules, the output should be -1. This problem is an algorithmic challenge that

2. It represents a number in binary that is a power of 5.

## cannot rearrange characters and must remain contiguous). Furthermore, these substrings need to be valid binary representations of

powers of 5 without leading zeros. The concept of dynamic programming may immediately come to mind to handle the optimization aspect of the problem—minimizing the number of substrings. Specifically, we use a depth-first search (DFS) with memoization to ensure efficient computation by

Here's how we can approach the solution: 1. Preprocess the binary equivalents of all possible powers of 5 that can be represented within the length of s and store them in a hash set ss. This preprocessing speeds up the checks needed later by allowing for constant-time lookups to see if a binary

avoiding redundant calculations.

- needed, and we return 0. o Otherwise, we iterate from the current index i to the end of the string, treating each possible end index j as the end of a candidate substring. We calculate the decimal value of the substring as we extend it and check if that value is in ss. If it is,
- 3. Optimize by using memoization to cache results of dfs(i) for each index i. This prevents the algorithm from re-computing the
- minimum number of substrings for any starting index more than once. 4. Call dfs(0) for the start of the string and if the value is infinite, we return -1 since it's impossible to partition the string into
- A recursive function dfs(i) is defined that takes an index i and returns the minimum number of beautiful substrings starting from this index. The recursion uses two crucial base conditions:

substring, it performs the following steps: It uses bit shifting to calculate the binary to decimal conversion of the substring s[i...j] while iterating.

 It continues to compute and track the minimum number of beautiful substrings (cuts) as it goes along. The memoization is achieved using Python's @cache decorator on the dfs function. This optimization ensures that once a substring

starting at index i is processed, the result is stored and thus any subsequent calls to dfs(i) will simply retrieve the stored result

At the end of recursion, if dfs(0) returns infinity (inf), it means the string s cannot be divided into beautiful substrings and thus the

loop that starts with x = 1 and keeps multiplying x by 5, adding each new power of 5 into the ss. This loop runs as long as the numbers being added are within the possible range dictated by the length of the string s.

1. Preprocess powers of 5 in binary: We first create a set ss of all binary strings that represent powers of 5 and are less than or equal to the length of s. Here is a small set for illustration: ss = {"1", "101", "10001", ...}. 2. Start with the recursive dfs function at index 0: We call dfs(0) and begin to explore all substrings starting from index 0.

Let's consider a small example to illustrate the solution approach step by step. Suppose we have the binary string s = "101101101"

 We can't start a substring with '0', so dfs(1) immediately moves on to dfs(2). • At index 2, we start with substring "1" again which is valid, and thus proceed to dfs(3). • And so on, until dfs(6) considers the substring "101" which is also a power of 5.

4. Using Memoization: Assume dfs(7) is called, and the results for this index are calculated and stored. If dfs(7) is called again

returns 0. • The result of dfs(9) is added to those of dfs(6), dfs(3), dfs(2), and dfs(0) to find the minimum number of beautiful

Therefore, for this example, our algorithm would successfully partition the string into the substrings {"1", "1", "1", "101"}, each of which

is a binary representation of a power of 5, and since we've used 4 substrings, the output would be 4. If a partition was not found, our

from functools import lru\_cache # Import the lru\_cache decorator for memoization class Solution:

# We cannot start with a '0' for a beautiful binary\_string

# Shift current\_value by one bit and add the new bit

# Check if current\_value is a power of 5

# And then add 1 for the current one

# Generate powers of 5 up to the length of the binary\_string

length = len(binary\_string) # Total length of the binary\_string

if current\_value in powers\_of\_five:

current\_value = (current\_value << 1) | int(binary\_string[j])</pre>

# Return -1 if there's no valid partition, otherwise the minimum substrings

# Calculate the minimum substrings if we take current substring

best\_result = min(best\_result, 1 + min\_substrings\_from\_index(j + 1))

number of beautiful substrings returned is 4 which are "1", "1", "1", and "101".

if binary\_string[index] == "0": return float('inf') # Represents an impossible scenario current\_value = 0 best\_result = float('inf') # Iterate over the binary\_string starting from current index

# If we have reached the end of the binary\_string, no more substrings needed

#### public class Solution { private Integer[] memoization; // memoization array to store results of subproblems private String inputString; // the input string 6 private Set<Long> powersOfFive; // set containing powers of five 8 private int inputLength; // the length of the input string

Java Solution

import java.util.HashSet;

import java.util.Set;

```
24
             // Attempt to find the minimum beautiful substrings starting from the first character
 25
             int result = findMinimum(0);
 26
 27
             // If the result exceeds the length of the input string, return -1, indicating no such decomposition
 28
             return result > inputLength ? -1 : result;
 29
 30
 31
         // Helper method to recursively find the minimum number of beautiful substrings starting at index i
 32
         private int findMinimum(int i) {
 33
             if (i >= inputLength) { // base case: if we've reached the end of the string
 34
                 return 0;
 35
 36
             if (inputString.charAt(i) == '0') { // no substring starting with a '0' can be beautiful
 37
                 return inputLength + 1;
 38
             if (memoization[i] != null) { // return the precomputed value if available
 39
                 return memoization[i];
 40
 41
 42
 43
             long binaryValue = 0; // to store the numerical value of the substring in binary
 44
             int ans = inputLength + 1; // initialize the minimum with an upper bound
 45
 46
             // Loop to consider all substrings starting at 'i'
             for (int j = i; j < inputLength; ++j) {</pre>
 47
                 binaryValue = (binaryValue << 1) | (inputString.charAt(j) - '0'); // accumulate the binary value
 48
 49
                 if (powersOfFive.contains(binaryValue)) { // if the binary value is a power of five
 50
                     // Attempt to find the minimum starting at the next character and add 1 for the current substring
 51
                     ans = Math.min(ans, 1 + findMinimum(j + 1));
 52
 53
 54
 55
             // Store the result in the memoization array before returning
 56
             return memoization[i] = ans;
 57
 58 }
 59
C++ Solution
  1 #include <unordered_set>
  2 #include <string>
    #include <cstring>
    #include <functional>
    using namespace std;
    class Solution {
     public:
         // Function to calculate the minimum number of beautiful substrings.
         int minimumBeautifulSubstrings(string s) {
 10
             unordered_set<long long> beautifulNumbers;
 11
 12
             int n = s.size();
 13
             long long powerOfFive = 1;
             // Populate a set with powers of 5. These represent the "beautiful numbers" in binary.
 14
             for (int i = 0; i \le n; ++i) {
 15
                 beautifulNumbers.insert(powerOfFive);
```

### Typescript Solution function minimumBeautifulSubstrings(s: string): number { // Set to hold powers of 5 const powersOfFive: Set<number> = new Set(); const length0fS = s.length;

6

8

9

10

11

12

13

**}**;

```
14
 15
         // Helper function to perform a depth-first search
 16
         const depthFirstSearch = (startIndex: number): number => {
 17
             // Base case: If we've reached the end of the string, return 0
             if (startIndex === length0fS) {
 18
                 return 0;
 19
 21
             // If the current character is a '0', it cannot be beautiful, return large number
 22
             if (s[startIndex] === '0') {
 23
                 return lengthOfS + 1;
 24
 25
             // Use memoization to avoid recalculating
             if (minSubstrFromIndex[startIndex] !== -1) {
 26
 27
                 return minSubstrFromIndex[startIndex];
 28
 29
             // Initialize a large number for comparison
 30
             minSubstrFromIndex[startIndex] = length0fS + 1;
 31
 32
             // Look ahead in the string to find valid beautiful substrings
 33
             for (let endIndex = startIndex, binaryValue = 0; endIndex < lengthOfS; ++endIndex) {</pre>
                 // Incrementally construct the binary value represented by the substring
 34
                 binaryValue = (binaryValue << 1) | (s[endIndex] === '1' ? 1 : 0);</pre>
 35
 36
                 // Check if the current binary value is a power of 5
 37
                 if (powersOfFive.has(binaryValue)) {
 38
                     // If it is, update the minimum count and recurse
 39
                     minSubstrFromIndex[startIndex] = Math.min(minSubstrFromIndex[startIndex], 1 + depthFirstSearch(endIndex + 1));
 40
 41
 42
 43
             // Return the minimum count of beautiful substrings starting from the current index
 44
             return minSubstrFromIndex[startIndex];
 45
         };
 46
 47
         // Start the depth-first search from the first character
 48
         const answer = depthFirstSearch(0);
 49
         // If the answer is larger than the length of the string, no valid solution exists, return -1
 50
 51
         return answer > length0fS ? -1 : answer;
 52 }
 53
Time and Space Complexity
The given Python code defines a recursive function dfs to compute the minimum number of beautiful substrings. The time and space
complexity analysis are as follows:
Time Complexity
The time complexity of this code is O(n^2). Here's the detailed reasoning:
```

## $(n-1) + \dots + 1$ , which has a sum of (n(n+1))/2, resulting in $0(n^2)$ time complexity. • Each recursive call involves a constant time check and a loop which shifts and adds digits to x, but these operations are 0(1) for each individual call. • The calls to min and to check if x in ss are also 0(1) since checking membership in a set is constant time on average.

**Space Complexity** The space complexity of this code is O(n) for the following reasons:

- The recursive function dfs could at most be called n times consecutively before reaching the base case (a call stack of depth n), thus the space complexity due to recursion is O(n).
- There's a cache used to store the results of subproblems in dfs due to the @cache decorator. The number of distinct subproblems is proportional to the length of the string s, also leading to a space complexity of O(n).

Therefore, the overall space complexity is dominated by these factors, resulting in O(n).

requires identifying the specific partitions that ensure the minimum count and conform to the rules. To solve this problem, it's crucial to recognize that we're not simply looking for subsequences, but specifically for substrings (which

Intuition

substring is a power of 5.

2. Implement the dfs function which operates recursively: ∘ If at an index i, where s[i] is '0', we can immediately return infinity (inf) because we can't have a leading zero.  $\circ$  If we reach the end of the string (i >= n), it means we have considered all characters, hence, no further partitions are

we have found a beautiful substring and we add 1 to the result of dfs(j + 1).

beautiful substrings. Otherwise, the value of dfs(0) gives us the minimum count of beautiful substrings. This leads us to a recursive solution with memoization that efficiently computes the minimum required partitions by only considering valid power of 5 numbers and by avoiding redundant checks through memoization.

Solution Approach The implementation starts with the creation of a set ss which contains the binary representation of all the numbers that are powers of 5 up to the maximum length of the binary string s provided. This set is pivotal for quickly verifying whether a substring can be

 When the current position i is at the end of the string (i >= n), which by definition means no further cuts are possible and it returns 0. • If the current bit is 0, indicating a leading zero if it were to be the start of a substring, it returns infinity to signify this cannot form a beautiful substring. The algorithm then progresses by considering all possible substrings starting from i up to the end of the string. For each candidate

considered beautiful.

 It checks if the current value, as it accumulates with each bit, exists in the precalculated powers of 5, stored in ss. If it does, it means the substring is beautiful. It calls dfs(j + 1) for the remainder of the string starting from j + 1, where j is current end of the considered substring. To this dfs call, 1 is added representing the current beautiful substring just processed.

function returns -1. Otherwise, the minimum number of cuts will be returned. To avoid repeatedly constructing the set ss for powers of 5, it is precomputed once before the recursive calls. This is performed in a Finally, the recursive function is initiated with dfs(0) to solve the entire string and the answer is checked against inf to return either

Example Walkthrough

instead of recalculating, reducing the time complexity significantly.

the minimum cuts or -1 if the partitioning isn't possible.

which we want to partition into beautiful substrings following the given conditions.

during the recursive calls, we don't recompute it but rather retrieve the stored result.

∘ "01101101" at dfs(1) skips the zero and then proceeds with "1" and call dfs(2).

"101101101" at dfs(0) would proceed with "1" and call dfs(1).

"1101101" at dfs(2) proceeds with "1" and call dfs(3).

"101101" at dfs(3) then finds "101", calls dfs(6).

5. Recursive and Memoization Results: Continues until the entire string is processed. During this process:

3. Exploring substrings: • We consider the first substring which is "1". Since "1" is in the ss set, it represents a power of 5. We then call dfs(1) (which is the next index) to explore further the rest of the string "01101101".

o "101" at dfs(6) is a power of 5 itself, so it calls dfs(9), now at the base case because the end of the string is reached and it substrings. For this example, dfs(9) returns 0, dfs(6) returns 1, dfs(3) returns 2, dfs(2) returns 3, and dfs(0) finally returns 4. 6. End of Recursion: After applying the memoization and the entire recursion, if the returned value is infinity (inf), our function outputs -1, indicating that it's not possible to partition s according to the rules. However, for s = "101101101", the minimum

algorithm would return -1.

Python Solution

if index >= length:

return best\_result

for i in range(length):

power\_of\_five \*= 5

inputLength = s.length();

powersOfFive = new HashSet<>();

powersOfFive.add(power);

memoization = new Integer[inputLength];

for (int i = 0; i <= inputLength; ++i) {</pre>

// Precompute powers of 5 and add to the set

this.inputString = s;

long power = 1;

power \*= 5;

powerOfFive \*= 5;

long long num = 0;

int minSubstrings[n];

// Array to store minimum beautiful substrings starting at each index

// Lambda function to calculate minimum beautiful substrings using DFS

return 0; // Base case: no more substrings, so return 0

if (s[idx] == '0') { // Beautiful substrings cannot start with '0'

return n + 1; // Return a big number which will not be minimum

num = (num << 1) | (s[j] - '0'); // Convert binary to decimal</pre>

return minSubstrings[idx] = ans; // Memoize and return the answer

if (beautifulNumbers.count(num)) { // If it's a beautiful number

// Take minimum of current answer and 1 plus the answer from next index

return ans > n? -1: ans; // If answer is greater than n, no beautiful substring is found, return -1

if (idx >= n) { // If the entire string has been processed

if  $(minSubstrings[idx] != -1) { // Check if already computed}$ 

int ans = n + 1; // Initialize the answer with a large number

ans = min(ans, 1 + dfs(j + 1));

// Array to hold the minimum beautiful substrings from each index

const minSubstrFromIndex: number[] = new Array(lengthOfS).fill(-1);

memset(minSubstrings, -1, sizeof(minSubstrings));

function<int(int)> dfs = [&](int idx) {

return minSubstrings[idx];

for (int j = idx; j < n; ++j) {

int ans = dfs(0); // Start DFS from index 0

// Pre-calculate powers of 5 and store in the set

for (let i = 0, power = 1; i <= length0fS; ++i) {</pre>

powersOfFive.add(power);

power \*= 5;

power\_of\_five = 1

for j in range(index, length):

# A set to store the powers of 5 values

powers\_of\_five.add(power\_of\_five)

return -1 if result == float('inf') else result

result = min\_substrings\_from\_index(0)

powers\_of\_five = {power\_of\_five}

# Start the search from index 0

return 0

def minimumBeautifulSubstrings(self, binary\_string: str) -> int: # A decorator that caches the return values of the function it decorates @lru\_cache(maxsize=None) def min\_substrings\_from\_index(index: int) -> int: 8 9 10 11 12 13 14 15 16 17

18

19

20

21

22

23

24

25

26

9 10 // Method to calculate the minimum number of beautiful substrings 11 public int minimumBeautifulSubstrings(String s) { 12 13 14 15 16 17 18

19

20

21

22

23

16 17 18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

• The function dfs is called recursively and it iterates over the entire length of the string s for each starting index i. In the worstcase scenario, the inner loop could run through the remaining part of the string, which gives us n iterations when starting at the first character, n-1 on the second, down to 1 iteration at the last character. Summing these up gives us the arithmetic series n +

• The hash set ss which contains at most n elements also requires O(n) space.