254. Factor Combinations

Medium Array Backtracking

Problem Description

The problem requires finding all unique combinations of an integer `n`'s factors, excluding `1` and `n` itself. A number's factors are the numbers that divide it evenly without leaving a remainder. For example, for `n = 8`, the combinations of factors can be `[2, 2, 2]` which multiply together to give `8`, or `[2, 4]` as `2 * 4 = 8` as well. The goal is to list all such combinations for any given `n`.

The constraints are that the factors must be between `2` and `n-1` inclusive, since `1` and `n` are not considered in this problem. The solution should return all the possible combinations in any order.

Intuition

The intuition behind the solution is to use Depth First Search (DFS) to explore all possible combinations of factors. We start with the smallest possible factor (which is `2`) and recursively divide `n` by it to get a new number that we further factorize.

We keep a temporary list `t` to keep track of the current combination of factors. When `n` is divisible by a number `i`, we add `i` to our current combination and then perform a recursive call with `n` divided by `i`. This is because once we have chosen `i` as a factor, the next factor has to be `i` or greater to maintain an increasing order and avoid duplicates.

Whenever we reach a state in recursion where the number `n` cannot be further divided by factors greater than or equal to `i`, we add a combination of the collected factors along with the current `n` to our answer. We then backtrack by popping the last factor from our combination list and continue exploring other possibilities by incrementally increasing `i`.

The solution relies on the fact that factors of a number `n` will always be less than or equal to the square root of `n`. Thus, we only need to search for factors up to the square root of `n`, which optimizes the search process significantly.

By systematically exploring each factor and its multiples, the DFS approach ensures that we find all possible combinations of factors for the given integer `n`.

Solution Approach

Let's break down the provided solution code and explain how it implements the DFS strategy for factorizing the number `n`:

1. **Depth First Search (DFS)**: The `dfs()` function is a recursive function that is central to our solution. It takes two arguments, `n` (the number to be factorized) and `i` (the starting factor).
2. **Base Case**: Each time `dfs()` is called, it first checks if there's already a list of factors collected in `t`. If so, it appends the current `n` with the collected factors as a possible combination of factors into `ans`.
3. **Recursion and Factorization**: The function then enters a loop where it iterates through all potential factors starting from the smallest candidate `i`. It iterates only up to the square root of `n` since we know factors come in pairs, and for any pair that multiplies to `n`, one of the factors must be less than or equal to the square root of `n`.
4. **Divisibility Check**: For each candidate factor `j`, the function checks whether `j` is a factor of `n` by verifying if `n % j == 0`. If it is, `j` is a valid factor and is added to the temporary list `t`.
5. **Recursive Exploration of Further Factors**: A recursive call to `dfs(n // j, j)` is then made to explore further factorization with the reduced number `n // j`, and the process ensures that we don't consider any factors less than our current `j` to maintain uniqueness in combinations.
6. **Backtracking**: After the recursive call, which explores the subsequent factors, it is important to revert the changes for the next iteration. This is achieved by popping the last element from `t` (which was the current factor).
7. **Increment and Continue Search**: Before concluding the iteration, the current factor `j` is incremented to continue the search for the next potential factor.

The use of a temporary list `t` to store the current combination of factors allows for easy backtracking. By adding and removing factors as we dive deeper into the recursion tree or backtrack, we effectively build all possible factor combinations. The outer list `ans` collects all unique combinations that are generated during the recursive process.

The solution employs recursion effectively to explore different factor combinations, backtracking to undo decisions and continue the search, and pruning in the form of stopping the factor search at the square root of `n`. These principles, combined, provide a robust and efficient method to solve the given problem.

Example Walkthrough

Let's illustrate the solution approach with an example by factorizing the integer `n = 12`:

1. **Starting the DFS**: We call the `dfs()` function initially with `n = 12` and `i = 2`. The temporary list `t` is empty, and `ans` is the list where we'll store our combinations.
2. **Base Case and Recursion**: On the first call to `dfs()`, since `t` is empty, we skip adding anything to `ans`. Now, we start our loop with `i = 2` and iterate up to the square root of `n`.
3. **Divisibility Check for `i = 2`**: We check if `12` is divisible by `2`. It is, so we add `2` to our temporary list `t` which now contains `[2]`.
4. **Recursive Call with Reduced `n`**: We make a recursive call to `dfs(12 // 2, 2)`, which is the same as `dfs(6, 2)`. This represents factorizing 6 with the smallest factor still being 2.
5. **Continuing Recursion with `n = 6`**: With `n = 6`, `t = [2]`, we repeat the steps. `6` is divisible by `2`, so we add `2` to `t` and it becomes `[2, 2]`. Then we call `dfs(6 // 2, 2)` which is `dfs(3, 2)`.
6. **Terminating Condition for `n = 3`**: With `n = 3`, `t = [2, 2]`, the loop checks for factors from `2` up to the square root of `3`. Since `3` isn't divisible by `2` and no other factors exist between `2` and the square root of `3`, the base case appends `[2, 2, 3]` to `ans`, which is a valid combination.
7. **Backtracking**: The function backtracks by popping the last element of `t` and increasing `i`. So `t` reverts back to `[2]`, and now `i` becomes `3`.
8. **Increment and Continue Search**: The iteration with `i = 3` checks if `12` is divisible by `3`. It is, so we add `3` to `t`, making it `[2, 3]`, and then we call `dfs(12 // 3, 3)`, which is `dfs(4, 3)`.
9. **Recursive Call with Reduced `n = 4`**: Now with `n = 4`, `t = [2, 3]`, we find `4` can't be further factorized with a starting factor of `3`, so the base case adds `[2, 3, 4]` to `ans`. Backtracking occurs again by popping the last element to continue with the next factor.

By following these steps, we would eventually explore all candidate factors for each recursive call to `dfs`, add valid combinations to `ans`, backtrack, and increment `i` to avoid repetitive combinations and ensure they are all unique.

The final `ans` list after exploring all possibilities and pruning through the use of recursion and backtracking would be `[[2, 2, 3], [2, 6], [3, 4]]`, which are all the unique combinations of factors (excluding `1` and `n` itself) that can multiply together to give `12`.

Python Solution

```python
from typing import List

class Solution:
    def getFactors(self, n: int) -> List[List[int]]:
        # Helper function to perform depth-first search
        def depth_first_search(target, start_factor):
            # If temp_factors has elements, then add a combination to the answer
            if temp_factors:
                answer.append(temp_factors + [target])
            # Initialize a factor to start from
            factor = start_factor
            # Check for factors only up to the square root of the target
            while factor * factor <= target:
                # If factor is a valid factor of target
                if target % factor == 0:
                    # Append the factor to the temporary list for possible answer
                    temp_factors.append(factor)
                    # Recurse with the reduced number (integer division)
                    depth_first_search(target // factor, factor)
                    # Pop the last factor to backtrack
                    temp_factors.pop()
                # Increment the factor
                factor += 1

        # A list to keep a temporary set of factors for a combination
        temp_factors = []
        # The final list of lists to be returned
        answer = []
        # Initiate depth-first search with the full target and the smallest factor
        depth_first_search(n, 2)
        return answer

```

Java Solution

```java
class Solution {
    private List<Integer> currentFactors = new ArrayList<>(); // A list to keep track of the current combination of factors
    private List<List<Integer>> allFactorCombinations = new ArrayList<>(); // A list to store all possible combinations of factors

    // This function initiates the process to find all unique combinations of factors (excluding 1 and the number itself) that multip
    public List<List<Integer>> getFactors(int n) {
        findFactors(n, 2);
        return allFactorCombinations; // Return the list of all factor combinations
    }

    // This recursive function finds all factor combinations for 'n' starting with the factor 'start'
    private void findFactors(int n, int start) {
        // If the currentFactors list is not empty, it means we have a valid combination of factors
        if (!currentFactors.isEmpty()) {
            List<Integer> combination = new ArrayList<>(currentFactors); // Make a copy of currentFactors
            combination.add(n); // Add the remaining 'n' to the combination
            allFactorCombinations.add(combination); // Add the new combination to the list of all combinations
        }

        for (int j = start; j <= n / j; ++j) { // We only need to check factors up to sqrt(n)
            if (n % j == 0) { // Check if 'j' is a factor of 'n'
                currentFactors.add(j); // Add 'j' to current combination
                findFactors(n / j, j); // Recursively find factors of n/j starting with 'j'
                currentFactors.remove(currentFactors.size() - 1); // Backtrack: remove the last factor added before the next iteratic
            }
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>

class Solution {
public:
    // Main function to return all unique combinations of factors of a given number
    // excluding 1 and the number itself.
    std::vector<std::vector<int>> getFactors(int n) {
        std::vector<int> currentCombination;                    // Current combination of factors
        std::vector<std::vector<int>> allCombinations;          // All unique factor combinations

        // A recursive depth-first search function to find all factor combinations
        std::function<void(int, int)> dfs = [&](int remain, int startFactor) {
            // If current combination is not empty, add the remaining number as a factor
            // and save the factor combination to the result
            if (!currentCombination.empty()) {
                // Copy the current combination and append the remaining number
                std::vector<int> tempCombination = currentCombination;
                tempCombination.emplace_back(remain);
                // Add the new combination to the list of all combinations
                allCombinations.emplace_back(tempCombination);
            }

            // Iterate through possible factors starting from 'startFactor'
            for (int factor = startFactor; factor <= remain / factor; ++factor) {
                if (remain % factor == 0) {
                    // Factor is a valid divisor of the remaining number, include it in the combination
                    currentCombination.emplace_back(factor);
                    // Continue searching for next factors of the updated remaining number 'remain / factor'
                    dfs(remain / factor, factor);
                    // Backtrack: remove the last factor before the next iteration
                    currentCombination.pop_back();
                }
            }
        };

        // Start the depth-first search from factor 2
        dfs(n, 2);
        return allCombinations;
    }
};

```

Typescript Solution

```typescript
// Represents the current combination of factors
let currentCombination: number[] = [];
// Contains all the unique factor combinations
let allCombinations: number[][] = [];

// Recursive depth-first search function to find all factor combinations
const dfs = (remain: number, startFactor: number): void => {
  // If current combination is not empty, add the remaining number as a factor
  // and save the factor combination to the result
  if (currentCombination.length > 0) {
    // Copy the current combination and append the remaining number
    const tempCombination = [...currentCombination, remain];
    // Add the new combination to the list of all combinations
    allCombinations.push(tempCombination);
  }

  // Iterate through possible factors starting from 'startFactor'
  for (let factor = startFactor; factor * factor <= remain; ++factor) {
    if (remain % factor === 0) {
      // Factor is a valid divisor of the remaining number, include it in the combination
      currentCombination.push(factor);

      // Continue searching for next factors of the updated remaining number 'remain / factor'
      dfs(remain / factor, factor);

      // Backtrack: remove the last factor before the next iteration
      currentCombination.pop();
    }
  }
};

// Main function to return all unique combinations of factors of a given number
// excluding 1 and the number itself.
const getFactors = (n: number): number[][] => {
  // Clears combinations from any previous calls
  currentCombination = [];
  allCombinations = [];

  // Start the depth-first search from factor 2
  dfs(n, 2);

  return allCombinations;
};

// Example usage:
// const factors = getFactors(32);

```

Time and Space Complexity

Time Complexity

The given Python code performs a depth-first search to find all combinations of factors for a given number `n`. The time complexity of such algorithms can be difficult to determine precisely due to the nature of the recursive calls and the varying number of factors for different numbers. However, we can establish an upper bound.

The outer loop, starting with `j = i` and proceeding while `j * j <= n`, will iterate approximately `sqrt(n)` times in the worst case for each recursive stack frame since we're checking factors up to the square root of `n`. Each time a factor is found, a recursive call is made, and this can occur up to a depth where each factor is 2 in the worst case (the number is a power of 2), which would be `log(n)` recursive calls deep.

However, note that not all levels of the recursion will iterate `sqrt(n)` times since the value of `n` gets smaller with each successful division. Plus, not all numbers are powers of 2, so many will have fewer recursive calls. Due to these considerations, the time complexity of the algorithm is difficult to express as a standard time complexity notation but is roughly bounded by `O(sqrt(n) * log(n))`.

Space Complexity

The space complexity of the algorithm involves the space for the recursion stack and the space needed to store the temporary list `t` and the final answer `ans`.

- The depth of the recursion stack will be at most `O(log(n))` because, in the worst case, we're dividing the number by 2 each time we find a factor, which would lead to a maximum depth that corresponds with the base-2 logarithm of `n`.
- At each recursive call, we're storing a list of factors. The length of `t` could also go up to `O(log(n))` in the worst case when each factor is 2.
- The `ans` list can theoretically have a number of elements as large as the number of possible combinations of factors. In the worst case, the number of potential combinations might grow exponentially with the number of factors.

Considering all the factors above, the space complexity for storing `t` and `ans` could be `O(log(n))` and `O(m)` respectively, where `m` is the number of combinations of factors.

Thus, the space complexity of the algorithm is `O(m + log(n))`, with `m` potentially being quite large depending on the structure of the factors of `n`.