Problem 1096: Brace Expansion II

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Under the grammar given below, strings can represent a set of lowercase words. Let

R(expr)

denote the set of words the expression represents.

The grammar can best be understood through simple examples:

Single letters represent a singleton set containing that word.

R("a") = {"a"}

R("w") = {"w"}

When we take a comma-delimited list of two or more expressions, we take the union of possibilities.

R("{a,b,c}") = {"a","b","c"}

R("{{a,b},{b,c}}") = {"a","b","c"}

(notice the final set only contains each word at most once)

When we concatenate two expressions, we take the set of possible concatenations between two words where the first word comes from the first expression and the second word comes from the second expression.

R("{a,b}{c,d}") = {"ac","ad","bc","bd"}

R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg", "acefh"}

Formally, the three rules for our grammar:

For every lowercase letter

x

, we have

R(x) = {x}

.

For expressions

e

1

, e

2

, ... , e

k

with

k >= 2

, we have

R({e

1

, e

2

, ...}) = R(e

1

) ∪ R(e

2

) ∪ ...

For expressions

e

1

and

e

2

, we have

R(e

1

+ e

2

) = {a + b for (a, b) in R(e

1

) × R(e

2

)}

, where

+

denotes concatenation, and

×

denotes the cartesian product.

Given an expression representing a set of words under the given grammar, return

the sorted list of words that the expression represents

.

Example 1:

Input:

expression = "{a,b}{c,{d,e}}"

Output:

["ac","ad","ae","bc","bd","be"]

Example 2:

Input:

expression = "{{a,z},a{b,c},{ab,z}}"

Output:

["a","ab","ac","z"]

Explanation:

Each distinct word is written only once in the final answer.

Constraints:

1 <= expression.length <= 60

expression[i]

consists of

'{'

,

'}'

,

','

or lowercase English letters.

The given

expression

represents a set of words based on the grammar given in the description.

Code Snippets

C++:

```
class Solution {
public:
vector<string> braceExpansionII(string expression) {

}
};
```

Java:

```
class Solution {
public List<String> braceExpansionII(String expression) {

}
}
```

Python3:

```
class Solution:
def braceExpansionII(self, expression: str) -> List[str]:
```

Python:

```
class Solution(object):
def braceExpansionII(self, expression):
"""
:type expression: str
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {string} expression
* @return {string[]}
*/
var braceExpansionII = function(expression) {

};
```

TypeScript:

```
function braceExpansionII(expression: string): string[] {

};
```

C#:

```
public class Solution {
public IList<string> BraceExpansionII(string expression) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** braceExpansionII(char* expression, int* returnSize) {

}
```

Go:

```
func braceExpansionII(expression string) []string {

}
```

Kotlin:

```
class Solution {
fun braceExpansionII(expression: String): List<String> {

}
}
```

Swift:

```
class Solution {
func braceExpansionII(_ expression: String) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn brace_expansion_ii(expression: String) -> Vec<String> {

}
}
```

Ruby:

```
# @param {String} expression
# @return {String[]}
def brace_expansion_ii(expression)

end
```

PHP:

```
class Solution {

/**
* @param String $expression
* @return String[]
*/
function braceExpansionII($expression) {

}
}
```

Dart:

```
class Solution {
List<String> braceExpansionII(String expression) {

}
}
```

Scala:

```
object Solution {
def braceExpansionII(expression: String): List[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec brace_expansion_ii(expression :: String.t) :: [String.t]
def brace_expansion_ii(expression) do

end
end
```

Erlang:

```
-spec brace_expansion_ii(Expression :: unicode:unicode_binary()) ->
[unicode:unicode_binary()].
brace_expansion_ii(Expression) ->
.
```

Racket:

```
(define/contract (brace-expansion-ii expression)
(-> string? (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
```

```
vector<string> braceExpansionII(string expression) {

}
};
```

Java Solution:

```
/**
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public List<String> braceExpansionII(String expression) {

}
}
```

Python3 Solution:

```
"""
Problem: Brace Expansion II
Difficulty: Hard
Tags: string, hash, sort, search, stack

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def braceExpansionII(self, expression: str) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def braceExpansionII(self, expression):
"""
:type expression: str
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {string} expression
* @return {string[]}
*/
var braceExpansionII = function(expression) {

};
```

TypeScript Solution:

```
/**
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function braceExpansionII(expression: string): string[] {

};
```

C# Solution:

```
/*
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public IList<string> BraceExpansionII(string expression) {

}
}
```

C Solution:

```
/*
* Problem: Brace Expansion II
* Difficulty: Hard
* Tags: string, hash, sort, search, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** braceExpansionII(char* expression, int* returnSize) {

}
```

Go Solution:

```
// Problem: Brace Expansion II
// Difficulty: Hard
// Tags: string, hash, sort, search, stack
```

```
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func braceExpansionII(expression string) []string {

}
```

Kotlin Solution:

```
class Solution {
fun braceExpansionII(expression: String): List<String> {

}
}
```

Swift Solution:

```
class Solution {
func braceExpansionII(_ expression: String) -> [String] {

}
}
```

Rust Solution:

```
// Problem: Brace Expansion II
// Difficulty: Hard
// Tags: string, hash, sort, search, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn brace_expansion_ii(expression: String) -> Vec<String> {

}
}
```

Ruby Solution:

```
# @param {String} expression
# @return {String[]}
def brace_expansion_ii(expression)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $expression
* @return String[]
*/
function braceExpansionII($expression) {

}
}
```

Dart Solution:

```
class Solution {
List<String> braceExpansionII(String expression) {

}
}
```

Scala Solution:

```
object Solution {
def braceExpansionII(expression: String): List[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec brace_expansion_ii(expression :: String.t) :: [String.t]
def brace_expansion_ii(expression) do
```

```
end
end
```

Erlang Solution:

```
-spec brace_expansion_ii(Expression :: unicode:unicode_binary()) ->
[unicode:unicode_binary()].
brace_expansion_ii(Expression) ->
.
```

Racket Solution:

```
(define/contract (brace-expansion-ii expression)
(-> string? (listof string?))
)
```