Problem 118: Pascal's Triangle

Problem Information

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

Problem Description

Given an integer

numRows

, return the first numRows of

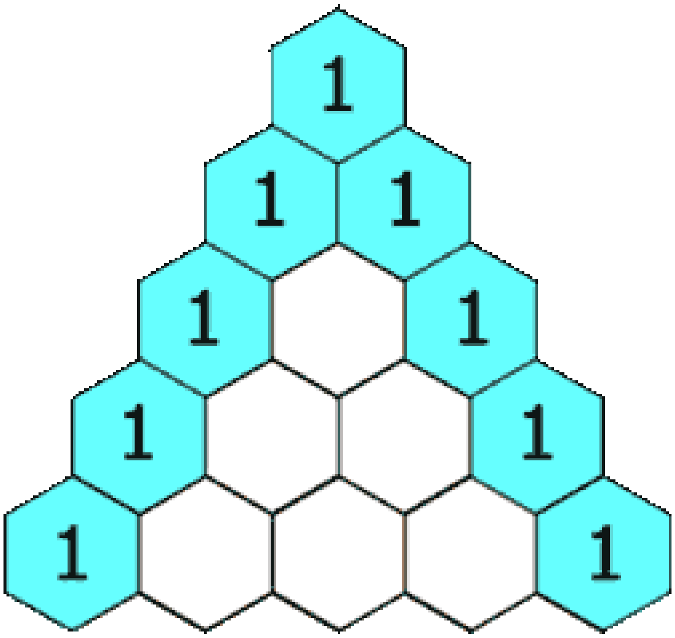
Pascal's triangle

.

In

Pascal's triangle

, each number is the sum of the two numbers directly above it as shown:



Example 1:

Input:

numRows = 5

Output:

[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

Example 2:

Input:

numRows = 1

Output:

[[1]]

Constraints:

1 <= numRows <= 30

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> generate(int numRows) {

}
};
```

Java:

```
class Solution {
public List<List<Integer>> generate(int numRows) {

}
}
```

Python3:

```
class Solution:
def generate(self, numRows: int) -> List[List[int]]:
```

Python:

```
class Solution(object):
def generate(self, numRows):
"""
:type numRows: int
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number} numRows
* @return {number[][]}
*/
var generate = function(numRows) {

};
```

TypeScript:

```
function generate(numRows: number): number[][] {

};
```

C#:

```
public class Solution {
public IList<IList<int>> Generate(int numRows) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** generate(int numRows, int* returnSize, int** returnColumnSizes) {

}
```

Go:

```
func generate(numRows int) [][]int {

}
```

Kotlin:

```
class Solution {
fun generate(numRows: Int): List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func generate(_ numRows: Int) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn generate(num_rows: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer} num_rows
# @return {Integer[][]}
def generate(num_rows)

end
```

PHP:

```
class Solution {

/**
* @param Integer $numRows
* @return Integer[][]
*/
function generate($numRows) {

}
}
```

Dart:

```
class Solution {
List<List<int>> generate(int numRows) {

}
}
```

Scala:

```
object Solution {
def generate(numRows: Int): List[List[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec generate(num_rows :: integer) :: [[integer]]
def generate(num_rows) do

end
end
```

Erlang:

```
-spec generate(NumRows :: integer()) -> [[integer()]].
generate(NumRows) ->
.
```

Racket:

```
(define/contract (generate numRows)
(-> exact-integer? (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
class Solution {
public:
vector<vector<int>> generate(int numRows) {

}
};
```

Java Solution:

```
/**
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public List<List<Integer>> generate(int numRows) {

}
}
```

Python3 Solution:

```
"""
Problem: Pascal's Triangle
Difficulty: Easy
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def generate(self, numRows: int) -> List[List[int]]:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

```
class Solution(object):
def generate(self, numRows):
"""
:type numRows: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} numRows
* @return {number[][]}
*/
var generate = function(numRows) {

};
```

TypeScript Solution:

```
/**
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

function generate(numRows: number): number[][] {

};
```

C# Solution:

```
/*
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public IList<IList<int>> Generate(int numRows) {

}
}
```

C Solution:

```
/*
* Problem: Pascal's Triangle
* Difficulty: Easy
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
```

```
*/
int** generate(int numRows, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Pascal's Triangle
// Difficulty: Easy
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func generate(numRows int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun generate(numRows: Int): List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func generate(_ numRows: Int) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Pascal's Triangle
// Difficulty: Easy
// Tags: array, dp
//
```

```
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn generate(num_rows: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer} num_rows
# @return {Integer[][]}
def generate(num_rows)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $numRows
* @return Integer[][]
*/
function generate($numRows) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> generate(int numRows) {

}
}
```

Scala Solution:

```
object Solution {
def generate(numRows: Int): List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec generate(num_rows :: integer) :: [[integer]]
def generate(num_rows) do

end
end
```

Erlang Solution:

```
-spec generate(NumRows :: integer()) -> [[integer()]].
generate(NumRows) ->
.
```

Racket Solution:

```
(define/contract (generate numRows)
(-> exact-integer? (listof (listof exact-integer?)))
)
```