Problem 1441: Build an Array With Stack Operations

Problem Information

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

Problem Description

You are given an integer array

target

and an integer

n

.

You have an empty stack with the two following operations:

"Push"

: pushes an integer to the top of the stack.

"Pop"

: removes the integer on the top of the stack.

You also have a stream of the integers in the range

[1, n]

.

Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to

target

. You should follow the following rules:

If the stream of the integers is not empty, pick the next integer from the stream and push it to the top of the stack.

If the stack is not empty, pop the integer at the top of the stack.

If, at any moment, the elements in the stack (from the bottom to the top) are equal to

target

, do not read new integers from the stream and do not do more operations on the stack.

Return

the stack operations needed to build

target

following the mentioned rules. If there are multiple valid answers, return

any of them

.

Example 1:

Input:

target = [1,3], n = 3

Output:

["Push","Push","Pop","Push"]

Explanation:

Initially the stack s is empty. The last element is the top of the stack. Read 1 from the stream and push it to the stack. s = [1]. Read 2 from the stream and push it to the stack. s = [1,2]. Pop the integer on the top of the stack. s = [1]. Read 3 from the stream and push it to the stack. s = [1,3].

Example 2:

Input:

target = [1,2,3], n = 3

Output:

["Push","Push","Push"]

Explanation:

Initially the stack s is empty. The last element is the top of the stack. Read 1 from the stream and push it to the stack. s = [1]. Read 2 from the stream and push it to the stack. s = [1,2]. Read 3 from the stream and push it to the stack. s = [1,2,3].

Example 3:

Input:

target = [1,2], n = 4

Output:

["Push","Push"]

Explanation:

Initially the stack s is empty. The last element is the top of the stack. Read 1 from the stream and push it to the stack. s = [1]. Read 2 from the stream and push it to the stack. s = [1,2]. Since the stack (from the bottom to the top) is equal to target, we stop the stack operations. The answers that read integer 3 from the stream are not accepted.

Constraints:

1 <= target.length <= 100

1 <= n <= 100

1 <= target[i] <= n

target

is strictly increasing.

Code Snippets

C++:

```
class Solution {
public:
vector<string> buildArray(vector<int>& target, int n) {

}
};
```

Java:

```
class Solution {
public List<String> buildArray(int[] target, int n) {

}
}
```

Python3:

```
class Solution:
def buildArray(self, target: List[int], n: int) -> List[str]:
```

Python:

```
class Solution(object):
def buildArray(self, target, n):
```

```
"""
:type target: List[int]
:type n: int
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {number[]} target
* @param {number} n
* @return {string[]}
*/
var buildArray = function(target, n) {

};
```

TypeScript:

```
function buildArray(target: number[], n: number): string[] {

};
```

C#:

```
public class Solution {
public IList<string> BuildArray(int[] target, int n) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** buildArray(int* target, int targetSize, int n, int* returnSize) {

}
```

Go:

```
func buildArray(target []int, n int) []string {

}
```

Kotlin:

```
class Solution {
fun buildArray(target: IntArray, n: Int): List<String> {

}
}
```

Swift:

```
class Solution {
func buildArray(_ target: [Int], _ n: Int) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn build_array(target: Vec<i32>, n: i32) -> Vec<String> {

}
}
```

Ruby:

```
# @param {Integer[]} target
# @param {Integer} n
# @return {String[]}
def build_array(target, n)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $target
```

```
* @param Integer $n
* @return String[]
*/
function buildArray($target, $n) {

}
}
```

Dart:

```
class Solution {
List<String> buildArray(List<int> target, int n) {

}
}
```

Scala:

```
object Solution {
def buildArray(target: Array[Int], n: Int): List[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec build_array(target :: [integer], n :: integer) :: [String.t]
def build_array(target, n) do

end
end
```

Erlang:

```
-spec build_array(Target :: [integer()], N :: integer()) ->
[unicode:unicode_binary()].
build_array(Target, N) ->
.
```

Racket:

```
(define/contract (build-array target n)
(-> (listof exact-integer?) exact-integer? (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<string> buildArray(vector<int>& target, int n) {

}
};
```

Java Solution:

```
/**
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public List<String> buildArray(int[] target, int n) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Build an Array With Stack Operations
Difficulty: Medium
Tags: array, stack

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def buildArray(self, target: List[int], n: int) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def buildArray(self, target, n):
"""
:type target: List[int]
:type n: int
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
/**
* @param {number[]} target
* @param {number} n
* @return {string[]}
*/
var buildArray = function(target, n) {

};
```

TypeScript Solution:

```
/**
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function buildArray(target: number[], n: number): string[] {

};
```

C# Solution:

```
/*
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public IList<string> BuildArray(int[] target, int n) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Build an Array With Stack Operations
* Difficulty: Medium
* Tags: array, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** buildArray(int* target, int targetSize, int n, int* returnSize) {

}
```

Go Solution:

```
// Problem: Build an Array With Stack Operations
// Difficulty: Medium
// Tags: array, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func buildArray(target []int, n int) []string {

}
```

Kotlin Solution:

```
class Solution {
fun buildArray(target: IntArray, n: Int): List<String> {

}
```

```
}
```

Swift Solution:

```
class Solution {
func buildArray(_ target: [Int], _ n: Int) -> [String] {

}
}
```

Rust Solution:

```
// Problem: Build an Array With Stack Operations
// Difficulty: Medium
// Tags: array, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn build_array(target: Vec<i32>, n: i32) -> Vec<String> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} target
# @param {Integer} n
# @return {String[]}
def build_array(target, n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $target
```

```
* @param Integer $n
* @return String[]
*/
function buildArray($target, $n) {

}
}
```

Dart Solution:

```
class Solution {
List<String> buildArray(List<int> target, int n) {

}
}
```

Scala Solution:

```
object Solution {
def buildArray(target: Array[Int], n: Int): List[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec build_array(target :: [integer], n :: integer) :: [String.t]
def build_array(target, n) do

end
end
```

Erlang Solution:

```
-spec build_array(Target :: [integer()], N :: integer()) ->
[unicode:unicode_binary()].
build_array(Target, N) ->
.
```

Racket Solution:

```
(define/contract (build-array target n)
(-> (listof exact-integer?) exact-integer? (listof string?))
)
```