969. Pancake Sorting

## **Problem Description**

Medium

The problem provides us with an array of integers arr. We are tasked with sorting this array but not through conventional means.

The only operation we can perform is what's termed as a "pancake flip." A pancake flip involves the following steps:

2. Reverse the sub-array arr[0...k-1], which is zero-indexed.

1. Select an integer k where 1 <= k <= arr.length.

Greedy Array Two Pointers Sorting

Our goal is to sort the entire array by performing a series of these pancake flips. To illustrate, if our array is [3,2,1,4] and we perform a pancake flip with k = 3, we reverse the sub-array [3,2,1], turning the array into [1,2,3,4]. The desired output is not the sorted array itself but rather a sequence of k values for each flip performed to sort the array. Importantly, any sequence that sorts the array within 10 \* arr. length flips is considered correct.

## The key to solving this problem lies in reducing it to smaller sub-problems. We can sort the array one number at a time,

Intuition

1. First, flip the largest number to the beginning of the array (unless it's already there). 2. Then flip it again to its final destination in the sorted array.

specifically by moving the largest number not yet in place to its final position through a sequence of at most two flips:

By repeating this process for the largest number down to the smallest, we can sort the entire array.

1. We iterate from the end of the array to the beginning (i = n-1 to 0), as the end of the array represents where the next largest number should

cross. No additional data structures are needed here; in-place swapping is sufficient.

go. 2. We then find the index j where the next largest number is (i+1 indicates the number we're looking for each iteration).

Now let's dive into the solution intuition:

- 3. If the largest number is not already in position i (j < i), we perform up to two flips: • The first flip moves the largest number to the start of the array (j = 0). This is only needed if the number is not already at the start (j > 0).
- The second flip moves the number from the start to its correct position i.
- We keep a list of k values for each flip performed and return it as the result.

A helper function named reverse is defined which takes the array and an index j. The function reverses the sub-array from

The main function, pancakeSort, iterates over the elements of the array in reverse order, from the last element down to the

The solution provided uses a simple greedy algorithm to sort the array using the constraints of pancake flips. To understand the implementation, let's walk through the algorithm and data structures used:

the start up to the j-th element (i.e., arr[0...j]). Simple swapping within the array is used for this operation. This is done using a while loop, where the i-th and j-th elements are swapped, incrementing i and decrementing j until they meet or

- second one (as the first one will be naturally sorted if all others are). Here, n is the length of the array. In each iteration, it finds the correct position j for the i-th largest value (which should be i+1 due to zero-indexing) by simply scanning the array from the start to the current position i. It uses a while loop for this, decrementing j until the value at
- arr[j] matches i+1. Once the position of the unsorted largest element is found, if it's not already at its correct position, we proceed with up to two
- value of j+1 is appended to the result list). • The next flip brings the element from the start to its designated position i by reversing the sub-array from the start up to i (a k value of i+1 is appended to the result list).

The algorithm keeps track of all flips performed by appending the k values to the ans list, which is the final result of the

The reasoning behind this approach is that each iteration ensures the next largest value is placed at its final position, reducing

o If j is greater than 0, which means that the element is not at the start, it flips the sub-array up to j to bring the element to the front (a k

- the problem's size by one each time. Since the relative order of the previous flips is not disrupted by further flips (they only affect elements before the largest placed element), the array is sorted correctly at the end of the algorithm.
- This approach does not require any advanced data structures. It leverages the fact that we can greedily place each largest unsorted value in its final place step by step, making it a simple yet powerful solution given the unique constraints of the problem.

1. Our goal is to sort the array [3,1,2] using pancake flips. 2. We start by looking for the position of the largest number that is not yet in its correct place. Since the array is of length 3, we begin with the number 3, which is also the largest.

3. We find that the largest number, 3, is already in the first position. This means we do not need the first flip; we can directly proceed to the

#### 4. We perform the second flip at the third position to move the number 3 to the end of the array. We reverse the sub-array [3,1,2], resulting in [2,1,3]. We add k=3 to our sequence of flips, as we flipped the first three elements.

illustration purposes.

**Python** 

second flip.

**Example Walkthrough** 

flips:

function.

5. Now, the largest number is in the correct position, and we ignore it in subsequent steps. We are left with considering the sub-array [2,1]. 6. Next, we move to the second largest number, which is 2. This is not in the correct position, so we find it at position 0.

9. Now, our array is fully sorted as [1,2,3], and our sequence of flips that were required is [3,2].

array [2,1], turning it into [1,2]. We add k=2 to our sequence of flips, as we flipped the first two elements.

Let's use a small example array [3,1,2] to illustrate the solution approach detailed above.

- 7. We perform our first flip to bring the number 2 to the beginning of the array (though it's already there, so this flip is redundant and can be skipped).
- 8. We then perform the second flip at the second position to move the number 2 to its correct location before the number 3. We reverse the sub-
- Thus, the sequence [3,2] represents the series of k values for each flip we performed to sort the array [3,1,2]. In practice, we could have avoided the redundant flip when the largest number was already at the beginning, but this walkthrough includes it for
- Solution Implementation
  - from typing import List class Solution: def pancakeSort(self, arr: List[int]) -> List[int]: # Helper function to reverse the array from start to index 'k'. def flip(arr, k):

#### k -= 1 n = len(arr)

start = 0

while start < k:</pre>

# Swap values

start += 1

arr[start], arr[k] = arr[k], arr[start]

# Find the index of the next largest element to sort

ans.add(i + 1); // Flip the sub-array from 0 to i

// Method to reverse the elements in the sub-array from index 0 to index j

int temp = arr[i]; // Temporary variable to hold the current element

return ans; // Return the list of flip positions

arr[i] = arr[j]; // Swap the elements

private void reverse(int[] arr, int j) {

arr[j] = temp;

for (int i = 0; i < j; i++, j--) {

if (arr[i] >= arr[maxIndex]) {

if (maxIndex == currentSize - 1) continue;

// the largest element to its correct position.

// Function to reverse the elements in the array from index 0 to end.

// Swap positions starting from the ends towards the center.

maxIndex = i;

reverse(arr, maxIndex);

reverse(arr, currentSize - 1);

output.push(currentSize);

output.push(maxIndex + 1);

return output; // Return the sequence of flips.

function reverse(nums: number[], end: number): void {

let start = 0; // Start of the sub-array to reverse.

if (maxIndex > 0) {

while (start < end) {</pre>

reverse(arr, i); // Performed to move the max element to the correct position

flips = [] # To store the sequence of flips

max\_index = arr.index(target\_index + 1)

# Start sorting from the end of the array

for target\_index in range(n - 1, 0, -1):

```
# If the largest element is not already in place
            if max_index != target_index:
                # Bring the largest element to the start if it's not already there
                if max_index > 0:
                    flips.append(max_index + 1)
                    flip(arr, max_index)
                # Now flip the largest element to its correct target index
                flips.append(target_index + 1)
                flip(arr, target_index)
       return flips
Java
import java.util.List;
import java.util.ArrayList;
class Solution {
   // Method to sort the array using pancake sort
   public List<Integer> pancakeSort(int[] arr) {
        int length = arr.length; // The length of the array
       List<Integer> ans = new ArrayList<>(); // The list to store the flips performed
       // Start from the end of the array and move towards the start
        for (int i = length - 1; i > 0; --i) {
           // Find the index of the next largest value expected at 'i'
           int maxIndex = i;
           while (maxIndex > 0 && arr[maxIndex] != i + 1) {
                --maxIndex;
           // Perform the necessary flips
            if (maxIndex < i) {</pre>
                // Flip the sub-array if the max index is not at the beginning
                if (maxIndex > 0) {
                    ans.add(maxIndex + 1); // Add the flip position to the answer list
                    reverse(arr, maxIndex); // Flip the sub-array from 0 to maxIndex
```

C++

```
#include <vector>
#include <algorithm>
class Solution {
public:
    std::vector<int> pancakeSort(std::vector<int>& arr) {
       int n = arr.size();
        std::vector<int> sorted_operations;
       // We iterate over each element in the array starting from the end
        for (int target_position = n - 1; target_position > 0; --target_position) {
            int current_index = target_position;
            // Find the index of the next largest element which should be at target_position
            for (; current_index > 0 && arr[current_index] != target_position + 1; --current_index);
           // If the element is already in the right place, skip it
            if (current_index == target_position) continue;
            // If the element is not at the beginning, flip the subarray so that the element
           // is at the beginning
            if (current_index > 0) {
                sorted_operations.push_back(current_index + 1); // We add +1 because indices are 1-based in pancake sort
                std::reverse(arr.begin(), arr.begin() + current_index + 1); // Perform the flip operation
            // Next, flip the subarray to move the element from the beginning to its target position
            sorted_operations.push_back(target_position + 1);
            std::reverse(arr.begin(), arr.begin() + target_position + 1);
       // Return the sequence of flips performed to sort the array
        return sorted_operations;
};
TypeScript
// Function to sort an array using pancake sort algorithm.
// The main idea is to do a series of pancake flips (reversing sub-arrays) to sort the array.
function pancakeSort(arr: number[]): number[] {
    let output = []; // This will store our sequence of flips.
    // Iterate over the array from the last element down to the second one.
    for (let currentSize = arr.length; currentSize > 1; currentSize--) {
        let maxIndex = 0;
       // Find the index of the largest element in the unsorted part of the array.
        for (let i = 1; i < currentSize; i++) {</pre>
```

```
start++;
          end--;
from typing import List
class Solution:
   def pancakeSort(self, arr: List[int]) -> List[int]:
        # Helper function to reverse the array from start to index 'k'.
        def flip(arr, k):
            start = 0
            while start < k:</pre>
                # Swap values
                arr[start], arr[k] = arr[k], arr[start]
                start += 1
                k -= 1
        n = len(arr)
        flips = [] # To store the sequence of flips
```

# Bring the largest element to the start if it's not already there

# Now flip the largest element to its correct target index

times in the worst case, which is when the largest element is at the beginning of the array.

[nums[start], nums[end]] = [nums[end], nums[start]]; // Perform the swap.

// If the largest element is already at its correct position, continue.

// Otherwise, flip the array at maxIndex and at currentSize — 1 to move

// Add the flip operations (+1 because operations are 1-based in pancakesort problem).

### The provided code achieves the goal of sorting a list through a series of pancake flips, which are represented as reversals of prefixes of the array. To analyze the time and space complexity, we need to consider both the number of reversals performed and the impact of each reversal on time and space.

the largest element at the end of the array.

# Start sorting from the end of the array

max\_index = arr.index(target\_index + 1)

# Find the index of the next largest element to sort

# If the largest element is not already in place

flips.append(max\_index + 1)

flip(arr, max\_index)

flips.append(target\_index + 1)

for target\_index in range(n - 1, 0, -1):

if max\_index != target\_index:

flip(arr, target\_index)

if max index > 0:

return flips

Time and Space Complexity

**Time Complexity:** 1. The outer loop runs from n - 1 down to 1, which gives us a total of n iterations. 2. Inside the outer loop, there is a while loop that searches for the position j of the i+1-th largest element. The while loop can iterate at most i

3. Two reversals can occur in each iteration of the outer loop; one if the largest element is not already in the right place, and another reversal puts

This yields a total of at most 2n reversals (each involving at most n elements to be swapped) across all iterations of the outer loop. Consequently, the worst-case time complexity is 0(n^2) because each of the 2n reversals takes up to 0(n) time.

# **Space Complexity:** The space complexity of the code consists of the space used by the input array and the additional space used to store the

answer list that keeps track of the flips. 1. No additional data structures that depend on the size of the input are allocated; the reversals are performed in place. 2. The ans list will contain at most 2n - 2 elements because, in the worst case, two flips are performed for each element except for the last one.

Therefore, the space complexity is O(n) for the ans list, in addition to O(1) auxiliary space for the variables used, which results in total space complexity of O(n).