1703. Minimum Adjacent Swaps for K Consecutive Ones Sliding Window Array Prefix Sum Greedy

Problem Description

Hard

swaps required to create a subarray of k consecutive 1s. The concept of a move involves selecting two adjacent positions in this array and swapping their values, with the restriction that a

In the given problem, we have an array nums which contains only us and 1s. Our goal is to find the minimum number of adjacent

Leetcode Link

move can only be made between two positions that are next to each other.

Intuition

The intuition behind solving this problem lies in focusing on the position of the 1s rather than the position of the 0s. Since we're only

interested in the position of 1s forming a consecutive sequence, we can abstract away the 0s by considering the original indices of

1s only.

We construct a new array, arr, to store the positions of the 1s from the original nums array. This abstracted view helps us in calculating the total moves without being influenced by the intermediating 0s. The procedure of the solution involves the following steps:

facilitate the calculation of moves. The prefix sum array s helps to calculate the total distance of 1s before and after a central

1. Record the positions of 1s: We iterate through nums and record the indices of 1s in a new list arr.

2. Calculate prefix sums: We find the prefix sums of the positions to have the accumulative distances in hand, which will later

point i.

3. The midpoint i in arr represents a hypothetical center of our sequence of k 1s, which means there are x 1s on its left and y 1s on

its right. We calculate x and y such that they represent the split of k 1s around the center i.

5. We repeat step 4 for all valid i and keep track of the minimum number of moves needed (ans).

- 4. For each valid midpoint 1, we calculate the minimum moves by considering the sum of distances from the left 1s (a) and right 1s (b) with adjustments for their positions relative to i.
- We use the concept of sliding window, which moves from left to right through our arr. Each iteration represents a potential solution where the x on the left and the y on the right are candidates for the required consecutive k 1s. Each potential solution provides the number of moves needed and we take the minimum of all these moves which is the answer.
- The above process enables us to calculate the minimum number of adjacent swaps to transform the nums array into one that has keeps are also as a second seco consecutive 1s in the least complex and most effective way.

The solution approach uses a sliding window technique, which is a common pattern used when dealing with subarrays or subsequences in an array. Let's walk through the code and understand the logic and use of algorithms and data structures. 1. First, we extract the positions of the 1s from the nums array and store them in a new list arr. This is achieved with a list

comprehension iterating over nums and including only indices where the value is 1. 1 arr = [i for i, x in enumerate(nums) if x == 1]

This step reduces the problem space from having to deal with both 0s and 1s to focusing solely on the positions of 1s, which

2. Next, we calculate the prefix sums of the positions stored in arr. We use accumulate from the itertools module, initializing the accumulation with (as implied by initial=0).

 $1 \times = (k + 1) // 2$

2 y = k - x

simplifies the problem significantly.

order to become a contiguous block.

Here k is the size of our target subarray:

1 for i in range(x - 1, len(arr) - y):

ls = s[i + 1] - s[i + 1 - x] rs = s[i + 1 + y] - s[i + 1]

Solution Approach

1 s = list(accumulate(arr, initial=0))

The prefix sum array s helps us quickly get the total distance that the 1s to the left and right of a mid-point have to travel in

3. For the calculations that follow, we need to determine the number of 1s on the left (x) and right (y) sides of our sliding window.

- This gives an odd k a left-heavier bias and puts the extra 1 on the left side. 4. We then iterate over the positions in arr where our target subarray of k 1s could be centered.
- Here i is the index in arr and j is the value at index i in arr, which corresponds to a position in the original nums array.

5. Inside the loop, we calculate the number of moves for the left partition (a) and the right partition (b). This is where the prefix

sums come in handy; they allow us to calculate the total cost to bring 1s on the left side of j to the left of j and the ones on the

After the loop completes, ans holds the answer to our original question — the minimum number of moves required so that nums has k

```
right side to the right, so that j would sit squarely in the middle of the k consecutive 1s.
6. Finally, we update and maintain the minimum number of moves required (ans) throughout our iterations:
```

consecutive 1s.

1 k = 3

10

2 x = (k + 1) // 2 -> x = 2

3 // Calculate left partition moves

7 // Calculate right partition moves

11 // Total moves for this midpoint

6. Next, for i = 2 (where arr[i] = 4):

3 // Calculate left partition moves

11 // Total moves for this midpoint

negative moves), our minimum is 1.

[1, 0, 1, 0, 1, 0, 1] is 1.

1 from itertools import accumulate

def min_moves(self, nums, k):

answer = float('inf')

x = (k + 1) // 2

y = k - x

Python Solution

class Solution:

6

8

9

10

11

12

13

14

15

16

17

18

19

20

26

27

28

29

30

33

34

37

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

43

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

41

40 }

42 }

1 j = arr[2] -> j = 4

12 ans = $a + b \rightarrow ans = -2 + 3 \rightarrow ans = 1$

3 y = k - x -> y = 1

1 ans = min(ans, a + b)

Example Walkthrough

1 nums = [1, 0, 1, 0, 1, 0, 1]

1. First, we identify the positions of all 1s in nums:

1 arr = [0, 2, 4, 6] // These indices in `nums` hold the value 1.

2. Next, we create a prefix sum array s based on the positions in arr:

Let's illustrate the solution approach using a small example. Suppose we have an array nums and a value of k = 3. Our nums array looks like this:

Our goal is to find the minimum number of adjacent swaps required to create a subarray of 3 consecutive 1s within nums.

1 s = [0, 0, 2, 6, 12] // With initial=0, prefix sums of `arr` with an additionally prepended 0. 3. We calculate x and y which will define the left and right partition sizes of k 1s around a midpoint index i in arr:

ls = s[i + 1] - s[i + 1 - x] -> ls = s[2] - s[0] -> ls = 6 - 0 -> ls = 6

8 rs = $s[i + 1 + y] - s[i + 1] \rightarrow rs = s[3] - s[2] \rightarrow rs = 12 - 6 \rightarrow rs = 6$

4 ls = s[i + 1] - s[i + 1 - x] -> ls = s[3] - s[1] -> ls = 12 - 0 -> ls = 12

Create an array of indices where the value is 1 (True)

Calculate the running sum of the indices array

for i in range(x - 1, len(indices_of_ones) - y):

// Collect indices of ones in the array

indicesOfOnes.add(i);

// Calculate prefix sums of indices

for (int i = 0; i < m; ++i) {

return (int) minMoves;

for (int i = 0; i < n; ++i) {

if (nums[i] == 1) {

Initialize the answer to be infinity

Calculate y as the larger half of k

prefix_sum = list(accumulate([0] + indices_of_ones))

indices_of_ones = [i for i, value in enumerate(nums) if value]

Calculate x as the smaller half of k (or equivalent for odd k)

Iterate over a range that is valid for a sliding window of size k

right_segment_sum = prefix_sum[i + 1 + y] - prefix_sum[i + 1]

Calculate the adjustments needed on the left side

Calculate the adjustments needed on the right side

int m = indicesOfOnes.size(); // Size of the subset array

prefixSums[i + 1] = prefixSums[i] + indicesOfOnes.get(i);

int leftHalf = (k + 1) / 2; // Number of elements in the left half

// Iterate through the array to find the minimum moves required

// Calculate the left and right cost for current index

// Update minMoves if the sum of costs is smaller

// Cast the long minMoves to int before returning

minMoves = Math.min(minMoves, leftCost + rightCost);

let minOperations: number; // Tracks the minimum number of operations required

prefixSum = new Array(totalOnes + 1).fill(0); // Initializing prefixSum array

// Calculate the group sizes around the middle element of the current window

// Sliding window over the array of positions of ones to find minimum moves

const sumLeft = prefixSum[i + 1] - prefixSum[i + 1 - leftGroupSize];

const sumRight = prefixSum[i + 1 + rightGroupSize] - prefixSum[i + 1];

// Update the minimum operations if the current operations count is less

minOperations = Math.min(minOperations, operationsForLeft + operationsForRight);

The given Python code comprises several parts whose time complexities will add up to the total time complexity.

2. Accumulate (s): The accumulate function constructs a prefix sum list, which also operates in O(n) time complexity.

iterates over all elements in nums to create a new list of indices arr where the elements are ones.

for(let i = leftGroupSize - 1; i < totalOnes - rightGroupSize; ++i) {</pre>

minOperations = Number.MAX_SAFE_INTEGER; // Setting minimum operations to max value

positionOfOnes = nums.map((val, index) => val === 1 ? index : -1).filter(index => index !== -1);

const rightGroupSize = k - leftGroupSize; // Number of elements to the right of the mid element

const current: number = positionOfOnes[i]; // The current position we are focusing on

const operationsForLeft = ((current + current - leftGroupSize + 1) * leftGroupSize / 2) - sumLeft;

const operationsForRight = sumRight - ((current + current + rightGroupSize) * rightGroupSize / 2);

const leftGroupSize = Math.floor((k + 1) / 2); // Number of elements to the left of mid element in the current window

// Function to find and return the minimum number of moves

// Extracting the positions of '1's from the input array

// Computing the prefix sums of the positions of ones

// Return the minimum number of operations found

totalOnes = positionOfOnes.length; // Calculating total ones

prefixSum[i + 1] = prefixSum[i] + positionOfOnes[i];

function minMoves(nums: number[], k: number): number {

for(let i = 0; i < totalOnes; ++i) {</pre>

for (int i = leftHalf - 1; i < m - rightHalf; ++i) {</pre>

int rightHalf = k - leftHalf; // Number of elements in the right half

long minMoves = Long.MAX_VALUE; // Initialize minMoves with a large value

int currentIndex = indicesOfOnes.get(i); // Current index in original array

int leftSum = prefixSums[i + 1] - prefixSums[i + 1 - leftHalf]; // Sum of left half

int rightSum = prefixSums[i + 1 + rightHalf] - prefixSums[i + 1]; // Sum of right half

long leftCost = (currentIndex + currentIndex - leftHalf + 1L) * leftHalf / 2 - leftSum;

long rightCost = rightSum - (currentIndex + 1L + currentIndex + rightHalf) * rightHalf / 2;

int[] prefixSums = new int[m + 1]; // Prefix sum array

Get the current central index around which we calculate the cost

left_adjustment = (central_index *2 - x + 1) $*x // 2 - left_segment_sum$

9 b = rs - (j + 1) * y -> b = 6 - (2 + 1) * 1 -> b = 6 - 3 -> b = 3

5 a = j * x - ls -> a = 2 * 2 - 6 -> a = 4 - 6 -> a = -2

 $5 \ a = j * x - ls -> a = 4 * 2 - 12 -> a = 8 - 12 -> a = -4$

We are looking for a subarray layout as 11[1]1 where the index i in arr would be the central [1].

5. When i = 1 (where arr[i] = 2): 1 j = arr[1] -> j = 2 (the index in `nums` corresponding to the midpoint `1`)

4. Now we iterate over the potential midpoints. The valid range of i is [x - 1, len(arr) - y], which in this case is [1, 2].

7 // Calculate right partition moves 8 rs = $s[i + 1 + y] - s[i + 1] \rightarrow rs = s[4] - s[3] \rightarrow rs = 12 - 12 \rightarrow rs = 0$ 9 b = rs - (j + 1) * y -> b = 0 - (4 + 1) * 1 -> b = 0 - 5 -> b = -510

12 ans = min(ans, a + b) -> ans = min(1, -4 - 5) -> ans = min(1, -9) -> ans = -9 (which can't be the case; ignored)

The negative values of a and b indicate that we have an overcount and must adjust our calculations accordingly.

7. The correct ans is the minimum of the total moves found while iterating. And as -9 isn't a feasible answer (we cannot have

Therefore, with k = 3, the minimum number of adjacent swaps required to create a subarray of 3 consecutive 1s in the array nums =

21 central_index = indices_of_ones[i] 22 23 # Calculate the left segment sum left_segment_sum = prefix_sum[i + 1] - prefix_sum[i + 1 - x] 24 25 # Calculate the right segment sum

31 right_adjustment = right_segment_sum - (central_index * 2 + y + 1) * y // 2 32 # Update the answer with the minimum of previous answer and the sum of adjustments answer = min(answer, left_adjustment + right_adjustment) 35 36 return answer # Return the minimum number of moves needed Java Solution class Solution { public int minMoves(int[] nums, int k) { 3 // Create a list to hold indices where nums[i] = 1 List<Integer> indicesOfOnes = new ArrayList<>(); 4 int n = nums.length; 5

```
1 class Solution {
2 public:
      int minMoves(vector<int>& nums, int k) {
          vector<int> positionOfOnes; // Holds the positions of '1's in the input vector.
```

C++ Solution

```
// Extracting the positions of '1's from the input vector.
  6
             for (int i = 0; i < nums.size(); ++i) {</pre>
                 if (nums[i] == 1) {
  8
                     positionOfOnes.push_back(i);
  9
 10
 11
 12
 13
             int totalOnes = positionOfOnes.size(); // The total number of '1's found.
             vector<long> prefixSum(totalOnes + 1, 0); // Prefix sum array for storing cumulative positions sum.
 14
 15
             // Computing the prefix sums of the positions of ones.
 16
             for (int i = 0; i < totalOnes; ++i) {</pre>
 17
                 prefixSum[i + 1] = prefixSum[i] + positionOfOnes[i];
 18
 19
 20
             long minOperations = LONG_MAX; // Initialize minimum operations to a large value.
 21
             int leftGroupSize = (k + 1) / 2; // Number of elements to the left of mid element in the current window.
 22
             int rightGroupSize = k - leftGroupSize; // Number of elements to the right
 23
 24
 25
             // Sliding window over the array of ones to find minimum moves.
 26
             for (int i = leftGroupSize - 1; i < totalOnes - rightGroupSize; ++i) {</pre>
                 int current = positionOfOnes[i]; // The current position we are focusing on.
 27
                 long sumLeft = prefixSum[i + 1] - prefixSum[i + 1 - leftGroupSize];
 28
                 long sumRight = prefixSum[i + 1 + rightGroupSize] - prefixSum[i + 1];
 29
 30
 31
                 long operationsForLeft = ((current + current - leftGroupSize + 1L) * leftGroupSize / 2) - sumLeft;
 32
                 long operationsForRight = sumRight - ((current + 1L + current + rightGroupSize) * rightGroupSize / 2);
 33
 34
                 // Update the minimum operations if the current moves are less.
 35
                 minOperations = min(minOperations, operationsForLeft + operationsForRight);
 36
 37
 38
             return minOperations; // Return the minimum number of operations found.
 39
 40 };
 41
Typescript Solution
  1 let positionOfOnes: number[] = []; // Holds the positions of '1's in the input vector
  2 let totalOnes: number; // The total number of '1's found
    let prefixSum: number[]; // Prefix sum array for storing cumulative positions sum
```

Time and Space Complexity **Time Complexity**

return minOperations;

3. Loop and Calculations within the Loop: The loop runs for approximately 0(n) iterations since it starts from x - 1 and goes until len(arr) - y. On each iteration, a constant number of arithmetic operations and array accesses are performed, which take 0(1)

Space Complexity

- The space complexity can be broken down as follows: 1. List arr: This list can contain at most n elements in the worst case where all elements in nums are ones. Therefore, the space
- 3. Variables ans, x, y, i, j, ls, rs, a, b: All of these variables use constant space, which is 0(1). When combining the space complexities, the dominating term is the space required for the lists arr and s, hence the overall space

2. Prefix Sum s: Similarly, the prefix sum list is of length n + 1, giving it a space complexity of O(n).

time each. Considering these parts together, the overall time complexity of the code is the sum of the individual complexities: 0(n) + 0(n) + O(n) * O(1). Simplified, it remains O(n).

1. List Comprehension (arr): This part has a time complexity of O(n), where n is the length of the input list nums. This is because it

complexity of the code is O(n).

complexity for this list is O(n).