1703. Minimum Adjacent Swaps for K Consecutive Ones

Problem Description

In the given problem, we have an array `nums` which contains only `0`s and `1`s. Our goal is to find the minimum number of adjacent swaps required to create a subarray of `k` consecutive `1`s.

The concept of a move involves selecting two adjacent positions in this array and swapping their values, with the restriction that a 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 `1`s rather than the position of the `0`s. Since we're only interested in the position of `1`s forming a consecutive sequence, we can abstract away the `0`s by considering the original indices of `1`s only.

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

The procedure of the solution involves the following steps:

1. Record the positions of `1`s: We iterate through `nums` and record the indices of `1`s 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 facilitate the calculation of moves. The prefix sum array `s` helps to calculate the total distance of `1`s before and after a central point `i`.
3. The midpoint `i` in `arr` represents a hypothetical center of our sequence of `k` `1`s, which means there are `x` `1`s on its left and `y` `1`s on its right. We calculate `x` and `y` such that they represent the split of `1`s around the center `i`.
4. For each central midpoint `i`, we calculate the moves by considering the sum of distances from the left `1`s (`L`) and right `1`s (`R`) with adjustments for their positions relative to `i`.
5. We repeat step 4 for all valid `i` and keep track of the minimum number of moves needed (`ans`).

We use the concept of sliding window, which moves from left to right through our `arr`. Each iteration represents a potential solution where the `i`-th the left and the `i`-th the right are candidates for the required consecutive `k` `1`s. 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 `k` consecutive `1`s in the least complex and most effective way.

Solution Approach

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 `1`s 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 `0`s and `1`s to focusing solely on the positions of `1`s, which simplifies the problem significantly.

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

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

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

3. For the calculations that follow, we need to determine the number of `1`s on the left (`x`) and right (`y`) sides of our sliding window. Here `i` is the size of our target subarray:

```
1 x = (k + 1) // 2
2 y = k - x
```

   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` `1`s could be centered.

```
1  for i in range(x - 1, len(arr) - y):
2      left = i - x + 1
3      lo = s[i + 1] - s[i + 1 - x]
4      ro = s[i + y + 1] - s[i + 1]
5      ans = min(ans, lo + ro)
```

   Here `i` is the index in `arr` and `lo` 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 (`lo`) and the right partition (`ro`). This is where the prefix sums come in handy; they allow us to calculate the total cost to bring `1`s on the left side of `i` to the left of `i` and the ones on the right side to the right, so that `i` would sit squarely in the middle of the `k` consecutive `1`s.
6. Finally, we update and maintain the minimum number of moves required (`ans`) throughout our iterations:

```
1 ans = min(ans, a + b)
```

   This step completes `ans` holds the answer to our original question — the minimum number of moves required so that `nums` has `k` consecutive `1`s.

Example Walkthrough

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:

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

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

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

```
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 `1`s around a midpoint index `i` in `arr`:

```
1 k = 3
2 x = (k + 1) // 2 // x = 2
3 y = k - x // y = 1
```

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

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]`.
5. When `i = 1` (where `arr[i] = 2`):

```
1 > x=1[] -> i - x + 1 (the index in `nums` corresponding to the midpoint `1`)
2
3 // Calculate left partition moves
4 lo = s[i + 1] - s[i + 1 - x] -> s[2] - s[0] -> 6 - 0 -> 6
5 a = j + x - 1 -> 4 + 2 + 2 - 1 -> 6 -> a = 6 - 0 -> 6 - 2
6
7 // Calculate right partition moves
8 ro = s[i + y + 1] - s[i + 1] -> s[3] - s[2] -> 12 - 6 -> 6
9 b = n - 1 - j + y -> 6 + 1 - 2 + 1 -> 6 - 5 -> 6 - 1 -> 5
10
11 // Total moves for this midpoint
12 ans = min(ans, a + b) -> ans = min(6, 5) -> ans = 5 (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.

6. The correct `ans` is the minimum of the total moves found while iterating. And as `-9` isn't a feasible answer (we cannot have negative moves), our minimum is `1`.

Therefore, with `k = 3`, the minimum number of adjacent swaps required to create a subarray of `3` consecutive `1`s in the array `nums = [1, 0, 1, 0, 1, 0, 1]` is `1`.

Python Solution

```
1 from itertools import accumulate
2
3 class Solution:
4     def min_moves(self, nums, k):
5         # Create a array of indices where the value is 1 (True)
6         arr = [i for i, x in enumerate(nums) if x == 1]
7         # Calculate the running sum of the indices array.
8         prefix_sum = list(accumulate(arr, initial=0))
9
10        # Initialize the answer to be infinity
11        answer = float('inf')
12
13        # Calculate x as the smaller half of k (or equivalent for odd k)
14        x = (k + 1) // 2
15        # Calculate y as the larger half of k
16        y = k - x
17
18        # Iterate over a range that is valid for a sliding window of size k
19        for i in range(x - 1, len(indices_of_ones) - y):
20            # Get the current central index around which we calculate the cost
21            central_index = indices_of_ones[i]
22
23            # Calculate the left segment sum
24            left_segment = prefix_sum[i + 1] - prefix_sum[i + 1 - x]
25            # Calculate the right segment sum
26            right_segment_sum = prefix_sum[i + 1 + y] - prefix_sum[i + 1]
27
28            # Calculate the adjustments needed on the left side
29            left_adjustment = central_index * x - (x * (x - 1)) // 2 - left_segment_sum
30            # Calculate the adjustments needed on the right side
31            right_adjustment = right_segment_sum - (central_index + 1) * y + (y * (y + 1)) // 2
32
33            # Update the answer with the minimum of previous answer and the sum of adjustments
34            answer = min(answer, left_adjustment + right_adjustment)
35
36        return answer  # Return the minimum number of moves needed
37
```

Java Solution

```
1  class Solution {
2      public int minMoves(int[] nums, int k) {
3          // Create a list to hold indices where nums[i] == 1
4          List<Integer> indicesOfOnes = new ArrayList<>();
5          int n = nums.length;
6
7          // Collect indices of ones in the array
8          for (int i = 0; i < n; ++i) {
9              if (nums[i] == 1) {
10                 indicesOfOnes.add(i);
11             }
12         }
13         int m = indicesOfOnes.size(); // Size of the subset array
14         int[] prefixSum = new int[m + 1]; // Prefix sum array
15
16         // Calculate prefix sums of indices
17         for (int i = 0; i < m; ++i) {
18             prefixSum[i + 1] = prefixSum[i] + indicesOfOnes.get(i);
19         }
20
21         long minMoves = Long.MAX_VALUE; // Initialize minMoves with a large value
22         int leftSubsize = (k + 1) / 2; // Number of elements to the left of mid
23         int rightSubsize = k - leftSubsize; // Number of elements to the right of mid
24
25         // Iterate through the array to find the minimum moves required
26         for (int i = leftSubsize - 1; i < m - rightSubsize; ++i) {
27             int currentIndex = indicesOfOnes.get(i); // Current index in original array
28             int leftSum = prefixSum[i + 1] - prefixSum[i + 1 - leftSubsize]; // Sum of left half
29             int rightSum = prefixSum[i + 1 + rightSubsize] - prefixSum[i + 1]; // Sum of right half
30
31             // Calculate the left and right cost for current index
32             long leftCost = (currentIndex * currentIndex - leftSubsize * leftSubsize) / 2 - leftSum;
33             long rightCost = rightSum - currentIndex * rightSubsize - rightSubsize * (rightSubsize + 1) / 2;
34
35             // Update minMoves if the sum of costs is smaller
36             minMoves = Math.min(minMoves, leftCost + rightCost);
37         }
38
39         // Cast the long minMoves to int before returning
40         return (int) minMoves;
41     }
42 }
```

C++ Solution

```
1  class Solution {
2  public:
3      int minMoves(vector<int>& nums, int k) {
4          vector<int> positionsOfOnes; // Holds the positions of '1's in the input vector.
5
6          // Extracting the positions of '1's from the input vector.
7          for (int i = 0; i < nums.size(); ++i) {
8              if (nums[i] == 1) {
9                  positionsOfOnes.push_back(i);
10             }
11         }
12
13         int totalOnes = positionsOfOnes.size(); // The total number of '1's found.
14         vector<long> prefixSumTotalOnes = 1, 0}; // Prefix sum array for storing cumulative positions sum.
15
16         // Computing the prefix sums of the positions of ones.
17         for (int i = 0; i < totalOnes; ++i) {
18             prefixSumTotalOnes[i + 1] = prefixSumTotalOnes[i] + positionsOfOnes[i];
19         }
20
21         long minOperations = LONG_MAX; // Initialize minimum operations to a large value.
22         int leftSubsquare = (k + 1) / 2; // Number of elements to the left of mid element in the current window.
23         int rightSubsquare = k - leftSubsquare; // Number of elements to the right of the mid element
24
25         // Sliding window over the array of ones to find minimum moves.
26         for (int i = leftSubsquare - 1; i < totalOnes - rightSubsquare; ++i) {
27             int current = positionsOfOnes[i]; // The current position we are focusing on.
28             long sumLeft = prefixSumTotalOnes[i + 1] - prefixSumTotalOnes[i + 1 - leftSubsquare];
29             long sumRight = prefixSumTotalOnes[i + 1 + rightSubsquare] - prefixSumTotalOnes[i + 1];
30
31             long operationsForLeft = (current * current - leftSubsquare * 1L) + leftSubsquare / 2) - sumLeft;
32             long operationsForRight = sumRight - (current * 1L + current + rightSubsquare) * rightSubsquare / 2;
33
34             // Update the minimum operations if the current operations count is less.
35             minOperations = min(minOperations, operationsForLeft + operationsForRight);
36         }
37
38         return minOperations; // Return the minimum number of operations found.
39     }
40 };
```

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
3  let prefixSum: number[] // Prefix sum array for storing cumulative positions sum
4  let minOperations: number; // Stores the minimum number of operations required
5
6  // Function to find and return the minimum number of moves
7  function minMoves(nums: number[], k: number): number {
8      // Extracting the positions of '1's from the input array
9      positionOfOnes = nums.map((val, idx) => val === 1 ? idx : -1).filter(index => index !== -1);
10
11     totalOnes = positionOfOnes.length; // Calculating total ones
12     prefixSum = new Array(totalOnes + 1).fill(0); // Initializing prefixSum array
13
14     // Computing the prefix sums of the positions of ones
15     for (let i = 0; i < totalOnes; i++) {
16         prefixSum[i + 1] = prefixSum[i] + positionOfOnes[i];
17     }
18
19     minOperations = Number.MAX_SAFE_INTEGER; // Setting minimum operations to max value
20
21     // Calculate the group sizes around the midpoint of the current window
22     const leftGroupSize = Math.floor((k + 1) / 2); // Number of elements to the left of mid element in the current window
23     const rightGroupSize = k - leftGroupSize; // Number of elements to the right of the mid element
24
25     // Sliding window over the array of positions of ones to find minimum moves
26     for (let i = leftGroupSize - 1; i < totalOnes - rightGroupSize; ++i) {
27         const current: number = positionOfOnes[i]; // The current position we are focusing on
28         const sumLeft = prefixSum[i + 1] - prefixSum[i + 1 - leftGroupSize];
29         const sumRight = prefixSum[i + 1 + rightGroupSize] - prefixSum[i + 1];
30
31         const operationsForLeft = ((current * current - leftGroupSize * 1) + leftGroupSize / 2) - sumLeft;
32         const operationsForRight = sumRight - (current * 1 + current + rightGroupSize) * rightGroupSize / 2);
33
34         // Update the minimum operations if the current operations count is less
35         minOperations = Math.min(minOperations, operationsForLeft + operationsForRight);
36     }
37
38     // Return the minimum number of operations found
39     return minOperations;
40 }
41
```

Time and Space Complexity

Time Complexity

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

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 iterates over all elements in `nums` to create a new list of indices `arr` where the elements are ones.
2. **Accumulate (`s`):** The `accumulate` function constructs a prefix sum list, which also operates in $O(n)$ time complexity.
3. **Loop and Calculations within the Loop:** The loop runs for approximately $O(n)$ 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 $O(1)$ time each.

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

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 complexity for this list is $O(n)$.
2. **Prefix Sum `s`:** Similarly, the prefix sum list is of length $n + 1$, giving it a space complexity of $O(n)$.
3. **Variables `ans`, `x`, `y`, `i`, `j`, `lo`, `ro`, `a`, `b`:** All of these variables use constant space, which is $O(1)$.

When combining the space complexities, the dominating term is the space required for the lists `arr` and `s`, hence the overall space complexity of the code is $O(n)$.