2786. Visit Array Positions to Maximize Score

```
Medium
                 Dynamic Programming
          <u>Array</u>
```

**Problem Description** 

You are provided with an int array nums that is 0-indexed and a positive int x. The goal is to calculate the maximum total score you can obtain starting at position 0 of the array and moving to any subsequent position. The rules are outlined as follows:

• You can move from your current position i to any position j such that i < j. • When you visit position i, you earn nums[i] points added to your score.

• Iterate through the nums array starting from index 1. For each value v at index i:

placeholder for "not yet computed or improbably low score."

def maxScore(self, nums: List[int], x: int) -> int:

Staying on the same parity (f[v & 1] + v),

- If you move between positions i and j and nums[i] and nums[j] have different parities (one is odd, the other is even), you lose x points from
- your score. You kick off with nums [0] points, and you have to figure out the maximum score that can be achieved under these conditions.

Intuition

The task is to maximize the score while taking into account that moving between numbers of different parity comes with a penalty of x points. To do this, we need to use dynamic programming to track the highest scores while considering the parity of the current number.

• Initialize a list f with two elements, set to negative infinity [-inf, -inf]. This list will keep track of the max scores for even and odd indices. • Set the element of f corresponding to the parity of nums [0] (even or odd) to nums [0]. This represents the score starting at position 0.

Here's an outline of the approach:

- ∘ Calculate the maximum score when staying on the same parity (f[v & 1] + v) and when changing parity (f[v & 1 ^ 1] + v x). Update f[v & 1] with the highest score from the above step.
- After processing all elements, the maximum score will be the maximum element from f.
- The key intuition in this solution comes from recognizing that at any index i in the array, you have two scenarios to consider: 1. The last score came from an index with the same parity as i. In this case, you just add the current value to the previous score since no penalty is incurred.
- subtract the penalty x. This process will lead us to the highest possible score, taking into account the penalty for switching parities.

2. The last score came from an index with different parity. Here, you add the current value to the previous score from the opposite parity and

The implementation uses a dynamic programming approach to compute the maximum score. Here's a step-by-step walkthrough:

Firstly, the algorithm initializes a list f with two elements, [-inf, -inf]. This record is to keep track of the two possible

The algorithm then iterates through elements in nums starting from index 1. For each value v, it computes the two possible

The update for the score at parity v & 1 chooses whichever of these two possibilities gives a higher score. This is done by

states for our score related to parity: even (0) and odd (1). In Python, -inf denotes negative infinity which is a useful

## The first element of nums is factored into our initial state. Since we always start at position 0, f[nums[0] & 1] is set to

scenarios:

class Solution:

return max(f)

gets updated.

Solution Approach

nums [0]. The expression nums [0] & 1 will be 0 if nums [0] is even, and 1 if it is odd, so it determines the index of f that

the max function. After evaluating all elements in nums, the maximum score is the highest value in f, which can be obtained using Python's built-in max(f) function.

The code snippet provided succinctly translates this approach into a Python function as part of a Solution class:

traversal through all potential positions and parities, significantly reducing the complexity of the problem.

Starting at position 0, nums[0] = 4, which is even, so we update f[0] with the value of nums[0].

∘ Switching parity (f[v & 1 ^ 1] + v - x). The ^ operator is a bitwise XOR, which flips the bit, effectively getting us the other parity.

f = [-inf] \* 2f[nums[0] & 1] = nums[0]for v in nums[1:]:  $f[v \& 1] = max(f[v \& 1] + v, f[v \& 1 ^ 1] + v - x)$ 

Each iteration effectively represents a choice at every position i with a value v from nums: taking its score as part of the existing

parity sequence or starting a new sequence of the opposite parity with an x penalty. The algorithm dynamically keeps track of

the best choice by updating only the score of the relevant parity after each decision. This pattern avoids the need for recursive

```
Example Walkthrough
  Let's illustrate the solution approach with a small example:
  Suppose we have the array nums = [4, 5, 2, 7, 3] and the penalty x = 3.
  We initiate the variable f with two elements [-inf, -inf] to keep track of the max scores for the even (f[0]) and odd indices
```

## • If we stay with odd, f[1] would become 5 (since -inf + 5 is just 5), but there's a catch: we start from an even index so we must apply the

• f = [4, 9]

• f = [-inf, -inf]

• f = [4, -inf]

(f[1]).

• f = [8, 9]

With nums[3] = 7, which is odd:

• Staying odd, f[1] + 7 = 9 + 7 = 16.

Lastly, nums [4] = 3, which is odd:

Next, nums [2] = 2, which is even:

We move to nums[1] = 5, which is odd:

penalty x. The new score would be 5 - 3 = 2.

• Staying with even parity, f[0] + 2 = 4 + 2 = 6. • Switching to odd, f[1] + 2 - x = 9 + 2 - 3 = 8. The higher score is 8, so f[0] becomes 8.

• If we switch to even, f[0] + nums[1] would be 4 + 5 = 9. We take the max of both, which is 9, so we update f[1].

- Switching to even, f[0] + 7 x = 8 + 7 3 = 12. We take the max which is 16 and update f[1]. • f = [8, 16]
- Staying odd, f[1] + 3 = 16 + 3 = 19. • Switching to even, f[0] + 3 - x = 8 + 3 - 3 = 8. The max is 19, so f[1] remains 19.

Solution Implementation

from typing import List

• f = [8, 19]

**Python** 

C++

public:

#include <vector>

class Solution {

using namespace std;

import math

class Solution:

The entire process demonstrates the dynamic programming algorithm's effectiveness in computing the maximum score by considering the penalty for switching between even and odd numbers. Each decision is based on whether to continue the

The maximum score we can get is  $\max(f)$ , which is 19.

# The list is used to track the maximum scores for even and odd numbers separately max\_scores = [-math.inf, -math.inf] # The first number's score is determined based on its parity (even/odd)

max\_scores[nums[0] % 2] = nums[0]

for value in nums[1:]:

parity = value % 2

def maxScore(self, nums: List[int], x: int) -> int:

# and assigned as the initial score for that parity

# Update the score for the current parity

long[] maxScoreForOddEven = new long[2];

Arrays.fill(maxScoreForOddEven, -(1L << 60));

// numParity is 0 for even and 1 for odd.

maxScoreForOddEven[numParity] = Math.max(

// Return the maximum score among the two parities.

maxScoreForOddEven[numParity] + nums[i].

return Math.max(maxScoreForOddEven[0], maxScoreForOddEven[1]);

maxScoreForOddEven[nums[0] & 1] = nums[0];

for (int i = 1; i < nums.length; ++i) {</pre>

int numParity = nums[i] & 1;

#include <algorithm> // For max() function

long long maxScore(vector<int>& nums, int x) {

# Initialize a list with two elements representing negative infinity

# Iterate over the remaining numbers starting from the second element

# Determine the parity of the current number, 0 if even, 1 if odd

# or switching parity and applying the penalty/subtraction of x

# Return the maximum score between the even and odd parities

// Iterate over the array, starting from the second element.

# max() is choosing the greater value between continuing the same parity

// Initialize both entries with a verv small number to simulate negative infinity.

// The first number decides the initial maximum score for its parity (odd or even).

// Update the maximum score for the current parity (odd or even).

 $max_scores[parity] = max(max_scores[parity] + value, max_scores[parity <math>^{1}$ ] + value - x)

return max(max\_scores) Java class Solution { // Method to calculate the maximum score. public long maxScore(int[] nums, int x) { // Array f to store the current maximum score for odd and even indexed numbers.

maxScoreForOddEven[numParity  $^{1}$ ] + nums[i] - x // Case when adding the current number leads to change in parit

// with the penalty x.

// Case when adding the current number to the same parity.

sequence of the current parity or start a new one of the opposite parity with a penalty. The algorithm avoids the need to check

each path separately, instead of using a running tally that gets updated in each step, which is considerably more efficient.

```
// Define an infinite value for long long type
        const long long INF = 1LL << 60;</pre>
        // Create a vector to track the maximum scores for even and odd indices
        vector<long long> maxScores(2, -INF); // Initialized with -INF
        // Initialize the first element of the score according to whether it's even or odd
        maxScores[nums[0] & 1] = nums[0];
        // Calculate the number of elements
        int n = nums.size();
        // Loop over the elements starting from the second element
        for (int i = 1; i < n; ++i) {
            // Update the max score for the current parity (even/odd index) of the number
            // This is the maximum of either adding the current number to the existing
            // score of the same parity, or switching parity and subtracting the penalty x
            maxScores[nums[i] \& 1] = max(
                maxScores[nums[i] & 1] + nums[i].
                                                              // Same parity: add current number
                maxScores[(nums[i] & 1) ^1] + nums[i] - x // Opposite parity: switch parity and subtract x
            );
        // Return the maximum value of the two max scores
        return max(maxScores[0], maxScores[1]);
};
TypeScript
function maxScore(nums: number[], x: number): number {
    // Define a very large number to represent "infinity".
```

// Initialize an array 'scores' with two elements representing the max scores for even and odd indices.

// The updated score is the max of the current score for the same parity plus the current number,

```
# max() is choosing the greater value between continuing the same parity
   # or switching parity and applying the penalty/subtraction of x
   max scores[parity] = max(max scores[parity] + value, max scores[parity ^ 1] + value - x)
# Return the maximum score between the even and odd parities
```

from typing import List

import math

class Solution:

const INFINITY = 1 << 30;</pre>

scores[nums[0] &  $\mathbf{1}$ ] = nums[0];

for (let i = 1; i < nums.length; ++i) {</pre>

return Math.max(scores[0], scores[1]);

const isOdd = nums[i] & 1;

const scores: number[] = Array(2).fill(-INFINITY);

// For the first number, update the score based on it being even or odd.

// or the score for the other parity plus the current number minus x.

# Initialize a list with two elements representing negative infinity

# The first number's score is determined based on its parity (even/odd)

# Iterate over the remaining numbers starting from the second element

# Determine the parity of the current number, 0 if even, 1 if odd

scores[is0dd] = Math.max(scores[is0dd] + nums[i], scores[is0dd ^ 1] + nums[i] - x);

# The list is used to track the maximum scores for even and odd numbers separately

// Loop through the numbers starting from the second element.

// Return the maximum score between the even and odd indices.

def maxScore(self, nums: List[int], x: int) -> int:

# and assigned as the initial score for that parity

# Update the score for the current parity

max\_scores = [-math.inf, -math.inf]

 $max\_scores[nums[0] % 2] = nums[0]$ 

for value in nums[1:]:

return max(max\_scores)

Time and Space Complexity

parity = value % 2

// Update the score for the current parity (even or odd).

**Time Complexity** The time complexity of this code is determined by the number of operations performed in the for-loop that iterates through the nums list:

• Within the loop, a constant number of operations are executed: two bitwise AND operations, four direct accesses by index to the list f, up to

The given Python code snippet aims to calculate a certain "maximum score" by iterating through the input list "nums" and applying

some operations based on the elements' parity (odd or even) and a given integer x. To analyze the time and space complexity of

two max operations, and a few arithmetic operations. Since these operations inside the loop are all of constant time complexity, the overall time complexity of the loop is 0(n - 1).

• The for-loop runs (n - 1) times, as it starts from the second element in nums.

this code, let's consider n to be the length of the input list nums.

Simplifying this, we get: 0(n-1) = 0(n).

**Space Complexity** 

As for the space complexity:

• Variable v is a single integer that is used to iterate through nums, which is also 0(1) space.

Thus, the time complexity of the code is O(n).

• There are no other data structures or recursive calls that use additional space that scales with the input size.

• A new list f of fixed size 2 is created. This does not depend on the size of the input and is thus 0(1).

Therefore, the space complexity of the code is 0(1). Without a reference answer provided alongside the code, the analysis is based solely on the provided snippet.