Kick Start 2021 - Round B

Analysis: Longest Progression

Test Set 1

For any test case, if $\mathbf{N} = 2$, we can return 2 as the answer immediately, as any two numbers are trivially an arithmetic progression by themselves. For $\mathbf{N} \geq 3$, we can attempt a strategy that tries to improve it by considering every element of $\mathbf{A}$ as a possible starting point. For instance, let us try to start a longest progression beginning at $\mathbf{A_0}$. Let us call the first difference we encounter, $\mathbf{A_1} - \mathbf{A_0}$, d. d could be the common difference of a longest progression that starts from $\mathbf{A_0}$, or it could not be. We attempt both possibilities:

1. If it is, we just need to continue trying to extend our possible progression with $\mathbf{A_2}$, $\mathbf{A_3}$ and so on, checking that $\mathbf{A_2} - \mathbf{A_1} = d$, $\mathbf{A_3} - \mathbf{A_2} = d$, etc. Because we can change at most one number, the first time we find some k for which $\mathbf{A_k} - \mathbf{A_{k-1}} \neq d$, we fix $\mathbf{A_k}$ so that this expression does equal to d. We continue along the array afterwards until we reach the end or we find a second instance of k where $\mathbf{A_k} - \mathbf{A_{k-1}} \neq d$; at that point, we cannot go any further.
2. If it is not, the only possibility for a progression with length $>= 3$ starting from $\mathbf{A_0}$ is if the common difference is $(\mathbf{A_2} - \mathbf{A_0})/2$. In this case, we must set $\mathbf{A_1}$ such that $(\mathbf{A_1} - \mathbf{A_0}) = (\mathbf{A_2} - \mathbf{A_1})$. Hence, note that this is only possible if $\mathbf{A_2} - \mathbf{A_0}$ is an even number - if it is not, we can determine the longest progression starting at $\mathbf{A_0}$ under this scenario to simply be 2 (as we cannot extend it past the first two values of the array). Once we have done so, our common difference is set, and we continue trying to extend our progression starting from $\mathbf{A_3}$, $\mathbf{A_4}$, and so on, stopping at the first $\mathbf{A_k}$ where $(\mathbf{A_1} - \mathbf{A_0}) \neq (\mathbf{A_k} - \mathbf{A_{k-1}})$. At that point, the length of the longest progression starting at $\mathbf{A_0}$ under this scenario will be k.

Both possibilities 1 and 2 take $O(\mathbf{N})$ for a given starting point. Because we will try sequentially all possible starting points from $\mathbf{A_0}, \mathbf{A_1}, \ldots, \mathbf{A_{N-2}}$, finding the max value amongst all $\mathbf{N} - 1$ possible starting points with 2 possible scenarios each, our overall time complexity with this strategy will be $O(\mathbf{N}^2)$.

Test Set 2

Let's simplify the problem a little. The only thing that really matters in this situation is the common difference between two adjacent numbers, not the numbers themselves. That is, the length of the longest progression of $[a, b, c, d, e, f]$ is always going to be the same as that of $[a + k, b + k, c + k, d + k, e + k, f + k]$. So, we can create an array D where $D_i = \mathbf{A_{i+1}} - \mathbf{A_i}$. D_0 is hence $\mathbf{A_1} - \mathbf{A_0}$, and so on, until $D_{\mathbf{N}-2}$ which is $\mathbf{A_{N-1}} - \mathbf{A_{N-2}}$. For instance, given $\mathbf{A} = [1, 3, 4, 7, 9, 11]$, we find that $D = [2, 1, 3, 2, 2]$.

Now, we can break D into *chunks*. Each chunk is a contiguous portion of the array with identical values. For a given chunk, let us call this value its *common value*. For instance, for $D = [2, 1, 3, 2, 2]$, we can break this down as $[2], [1], [3], [2, 2]$, so 2 is the common value of the first chunk, and so on. Each chunk has a starting index within D. That index should be associated with the length and common value of that chunk, and this information stored for lookup. Then, we iterate through the chunks in ascending starting index order. For each chunk beginning at index i, with length k, and having common value d, we greedily try to fix the element in D right after the chunk, if it exists. Then, we also need a separate attempt to try fixing the element right before the chunk, if it exists. That means we will need to attempt merging for

each chunk up to two times. The following analysis is for what happens when we merge with the element *after* our current chunk, but the same applies for the corresponding situation when we merge backwards too.

The last element in our current chunk has index $(i + k - 1)$. If we greedily use our ability to change a number by *adding* some amount to D_{i+k} so it is equal to d, we must also reduce D_{i+k+1} by the same amount. The opposite case applies. At this point:

- if $D_{i+k} + D_{i+k+1} \neq 2 \times d$, then we have only been able to merge our current chunk with the element right after. The length of our extended chunk is now $k + 1$.
- if $D_{i+k} + D_{i+k+1} = 2 \times d$, but $D_{i+k+2} \neq d$, then changing D_{i+k} has allowed us to also merge with D_{i+k+1}. The length of our extended chunk is now $k + 2$.
- if $D_{i+k} + D_{i+k+1} = 2 \times d$, and $D_{i+k+2} = d$, then we have been able to merge even further with a chunk that starts 3 elements after our current one ends. The length of the whole merge becomes $k + 2+$ the length of the next chunk with common value d.

For instance, with $D = [1, 1], [2], [0, 0]$, if we try to merge forward from the first chunk which begins at index $i = 0$, has length $k = 2$ and common value $d = 1, D_2 + D_3 = 2 + 0 = 2 \times d$, but $D_4 \neq d$. This is the second scenario above, and the length of our extended chunk is now 4, so the overall progression length in $\mathbf{A}$ is 5. Another example: with $D = [2], [1], [3], [2, 2]$, if we try to merge forward from the first chunk which begins at index $i = 0$, has length $k = 1$ and common value $d = 2, D_1 + D_2 = 1 + 3 = 4 = 2 \times d$, and $D_3 = d$. This is the third scenario above, and the length of our extended chunk is now $1 + 2 + 2 = 5$, so the overall progression length in $\mathbf{A}$ is 6.

Our overall solution to the problem is then just the max of all the possible chunks made by considering each chunk in turn, and trying to merge it backwards and forwards. (As a result, if the test case in question falls into the third scenario listed above, the "longest progression" will actually be discovered twice, but this doesn't change its correctness.)

In summary, we run through the initial array once to create D, and then run through D to create map(s) of chunks tagged with the appropriate information, and then run through D again with our map of chunks to finally get our result. Assuming we have access to constant-time lookup for the lengths and common values of chunks, the overall time complexity is $O(\mathbf{N})$.