Solution

We begin by defining a few notions. Let the sequence of consecutive measurements be denoted by $a_1, a_2, \ldots, a_n$. We say that the segment $[i,j]$ is *t-stable* for a given t if it satisfies the requirement of keeping the measurements within the range t, so for each pair of indices i', j', such that $i \le i' \le j' \le j$, we have $|a_{i'} - a_{j'}| \le t$. Our task is to determine the longest t-stable segment for a given constant t.

Naïve $O(n^3)$ solution

The first idea is to consider all pairs of indices i, j and determine the one that corresponds to the longest t-stable segment.

```
record := 1 {the length of the longest t-stable segment so far}
for i := 1 to n do
    for j := i + 1 to n do
        if t-stable(i,j) then
            if j – i + 1 > record then
                record := j – i + 1
return record
```

How can we verify if a given segment $[i,j]$ is stable? Of course, we could consider all pairs i', j' such that $i \le i' \le j' \le j$, and verify if all values $a_{i'} - a_{j'}$ fit within the tolerance t. Such a solution would be too expensive: since there are quadratically many such pairs (i',j') for each of the $O(n^2)$ segments $[i,j]$, its complexity would be $O(n^4)$. But a simple observation enables us to test the t-stability of each of the $[i,j]$ segments in linear time. To assure t-stability, one has to know only two values: the minimum and the maximum within the segment. If the difference between them does not exceed t, then such a segment is t-stable; otherwise it is not. We can do it easily using $2n - 2$ comparisons, and with some optimization even $\lceil 3n/2 \rceil - 2$ comparisons are enough. So, having found this idea, we have obtained a solution with a cost of $O(n^3)$, but we are still far away from the optimal solution.

Pilots, by Piotr Chrząstowski

Improved $O(n^2)$ solution

The first important remark is that extending a non-t-stable segment from any end cannot make it t-stable. In other words, if the segment $[i,j]$ is not t-stable, then no segment $[i',j']$ is t-stable for any $i' \le i$ or $j \le j'$. This means that if, for the fixed beginning of the segment i, we are moving the end of the segment j forward, and recognize that for some j the segment $[i,j]$ is not t-stable, then we can stop regarding i as an interesting beginning: no further segments starting with i will be t-stable, and hence the record will not be broken. Note that when we go forward with j, we can update the maximal and minimal values if only a_j happens to be the biggest or the smallest of the considered values. This is the way to obtain a still-simple $O(n^2)$ algorithm.

```
i := 1
record := 1
while i ≤ n − record do
    {there is no need to consider greater i, because such i}
    {cannot be the beginning of the longest t-stable segment}
    min := a[i] {the smallest... }
    max := a[i] {and the biggest value in the range}
    j := i + 1
    stable := true
    while (j ≤ n) and stable do
        if a[j] > max then
            max := a[j]
        else if a[j] < min then
            min := a[j]
        if max − min > t then
            stable := false
        else
            j := j + 1
    if j − i > record then
        record := j − i
    {we need not check this condition at each loop iteration}
    {because we know the best j for given i}
return record
```

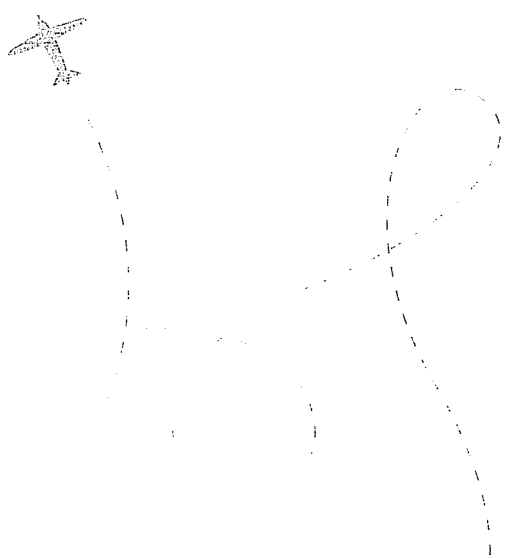
Note that we exit the loop with j as the smallest index for which the segment starting at i is not stable. The difference $j - i$ gives us the number of elements of the longest stable segment starting at i.

We have here one linear loop nested inside another, so the total cost of this solution is quadratic. Cutting the cost even further requires some more advanced techniques.

Still better $O(n \log n)$ solution

The main idea now is to increment two pointers representing the ends of a stable segment. If a_j destroys the stability then we move i forward; otherwise we increment j. In order to test the stability quickly, we should know the minimal and maximal values of the considered segment. To do it rapidly we will need a special technique to represent a multiset containing elements $a_i,\ldots,a_j$ in such a way that determining the minimal and maximal values is cheap. We need a multiset representation because the values can be repeated within one segment.

If we use balanced trees, like AVL, then with each value we will keep its multiplicity. The height of an AVL tree is logarithmic in the number of represented values. So, since the minimal and maximal values lie on the leftmost and the rightmost paths respectively, they can be determined in $O(\log n)$ time.



The pseudocode of this algorithm is quite simple.

```
i := 1
InitAVL(a[1]) {creates 1-element AVL tree containing a[1]}
record := 1
min := a[1]
max := a[1]
j := 2
while j ≤ n do
    InsertAVL(a[j])
    if a[j] > max then
        max := a[j]
    else if a[j] < min then
        min := a[j]
    while max − min > t do
        {if a[j] did not change min or max or if it did change, but}
        {without losing t-stability, then this loop will not be executed}
        DeleteAVL(a[i])
        Update(min, max)
        i := i + 1
    j := j + 1
    if j − i > record then
        record := j − i
return record
```

We can update the values *min* and *max* after deletion of $a[i]$ in a cleverer way by modifying the procedures *InsertAVL* and *DeleteAVL*, but let's not pursue this idea too deeply. Anyway, the cost of each of these operations is at most $O(\log n)$, even without this optimization.

The whole algorithm runs in $O(n \log n)$ time. The cost is so small because the number of loop iterations is linear: at least one of the indices i and j is incremented in each iteration (it does not matter that one of the loops is nested inside the other: together they can run at most $2n - 2$ times). In each loop iteration, every access to the multiset data structure requires $O(\log n)$ operations, thanks to the AVL trees implementation.

Linear solutions

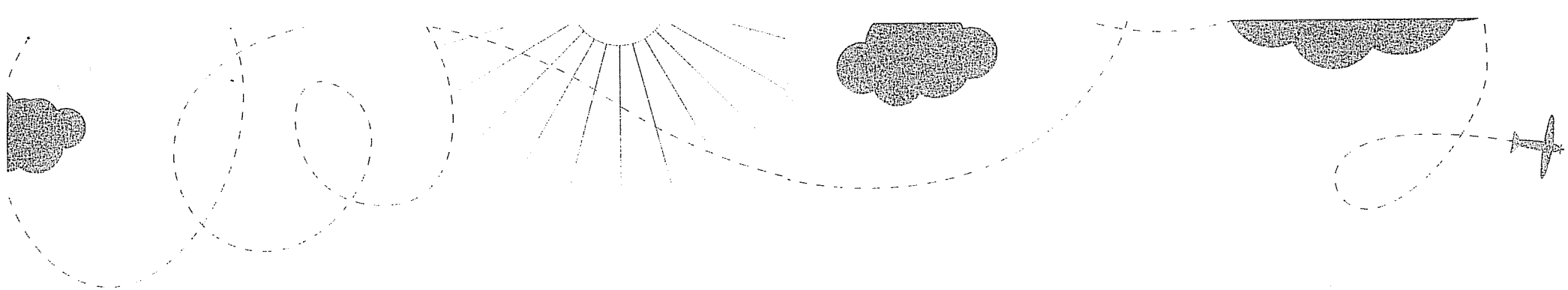
The key observation is that we do not need to remember all the measurements from the segment $[i,j]$, but only the minimum and maximum ones. If we are able to do it quickly, the rest of the values will not be needed. We are going to present two solutions, each with a linear time cost.

Linear-time solution Let $p[j]$ for each j denote the smallest index i for which the $[i,j]$ segment is t-stable. We are looking for the longest t-stable segment, so if we know $p[j]$ for each j, then comparing the values $j - p[j] + 1$ will give us the result. Let $minval(i,j)$ and $maxval(i,j)$ be the smallest and the biggest values in the segment $[i,j]$.

We will find the solution by using inductive reasoning: a powerful method that is present in many efficient algorithms. In our case, we will consider consecutive elements a_j for $j = 2,\ldots,n$ and design a method to advance from $j-1$ to j. Let's assume inductively that we know the value $p[j-1]$ and want to use this knowledge in determining $p[j]$. The first question is why this value should be different from $p[j-1]$. This can happen only when a_j destroys stability, so either $a[j] - minval(p[j-1],j-1) > t$, or $maxval(p[j-1],j-1) - a[j] > t$. When none of these inequalities holds, then simply $p[j] = p[j-1]$.

The case in which the stability is broken, so $p[j] \neq p[j-1]$, requires finding the smallest index i that makes the $[i,j]$ segment stable. Consider three cases:

- If $a[j-1] > a[j]$ then the danger comes only from the top. The new element $a[j]$ can be too small for some of the elements from the segment $[p[j-1],j-1]$. So we should find among these indices some k such that none of the elements $a[k],\ldots,a[j-1]$ exceeds $a[j]$ by more than t. It is hence convenient to know the indices of the decreasing left-maximums in this segment, beginning with the greatest value in the whole segment. (A left-maximum in a segment is an element such that none of the elements with a bigger index is greater.)
- If $a[j-1] < a[j]$ then the danger comes only from the bottom. Analogously, we should have easy access to the increasing sequence of left-minimums. (A left-minimum in a segment is an element such that none of the elements with a bigger index is smaller.)
- If $a[j] = a[j-1]$ then we have nothing to do, except for verifying that the current segment is the longest (we have $p[j] = p[j-1]$, and the segment is now longer).



We see here that the key to success is the information about two specific sequences: the decreasing sequence of left-maximums and the increasing sequence of left-minimums for the segment $[p[j-1],j-1]$. In each of these cases we will memorize the indices of the stored values. We should also decide which of the possibly equal values will be stored. It is quite clear: if some left-maximum value m is too big for a given $a[j]$, then all the occurrences of m in the segment $[p[j-1],j-1]$ are equally bad, so we will be interested in the last occurrence of each such value. The first candidate for $p[j]$ will be the element whose index is greater by 1 than the last occurrence of this left-maximum. It can happen, however, that consecutive left-maximums (even all of them) are too big. In such a case, we should stop at any such leftmost left-maximum that does not violate t-stability (in the extreme case, the considered element $a[j]$ can become such a left-maximum, so we will always find one). An analogous situation concerns left-minimums: we need the indices of the latest occurrences of increasing values being left-minimums. For instance, for $t = 4$ and a 4-stable sequence

5, 4, 6, 8, 7, 7, 4, 6, 4, 5

the interesting decreasing values of left-maximums are 8, 7, 6, 5 with the corresponding indices being 4, 6, 8, 10, while the increasing values of left-minimums are 4, 5 with indices 9, 10. Clearly here we have $p[10] = 1$. If we extend the considered sequence with an 11th element, we must modify these two sequences. For instance, augmenting our sequence by $a[11] = 2$ we have $p[11] = 7$—the longest 4-stable segment begins with the value 4 at the 7th position. The updated sequence of left-maximums for this new 4-stable segment ending on the added 2 will be equal to 6, 5, 2, with indices 8, 10, 11, while the sequence of left-minimums will have just one element: 2 with the index 11.

Note that the actual value of $p[j]$ is not important except for the case where we increment j without changing $p[j]$. Only then can the record be beaten.

Let us make it more formal. For each j we define the decreasing sequence of left-maximums UP_j in the following way. The first element of the sequence is the biggest value in the segment $[p[j],j]$. If there is more than one such value, we choose the last one. Consecutive elements of this sequence are the last occurrences of such elements x that are smaller than the previous chosen elements, but which satisfy the property that up to the end of the segment (so up to j) there are no bigger values than x. The increasing sequence of left-minimums $DOWN_j$ is defined analogously, starting from the latest occurrence of the smallest element in the segment $[p[j],j]$.

Having defined these two sequences, we can easily determine the value $p[j]$. When we look at the first sequence UP_j, we see that all the values that are bigger than $a[j]$ by more than t should be excluded, together with all the preceding elements. So a candidate for $p[j]$ will be the index bigger by 1 than the last element of UP_j, which destroys the t-stability. Of course, if even the first element of the sequence is not too big, then we stop this part of the investigation. In this case, $p[j]$ is possibly equal to $p[j-1]$. If, instead, the first element of the sequence exceeds $a[j]$ by more than t, then we delete from UP_j all elements that are too big and stop the deletion on such an element whose value is bigger than $a[j]$ by at most t. We memorize the index of the last deleted element. The next index is a candidate for $p[j]$. Note that it may happen that all the elements of UP_j will be deleted.

We perform the analogous procedure for the other sequence $DOWN_j$, determining another candidate for $p[j]$. Neither of them will object to a new segment's end $a[j]$, and both of them have indices greater by 1 than the last deleted elements. The value $p[j]$ (new value for i) is equal to the bigger index of these two candidates.

Now the time has come to update the values of *UP* and *DOWN* from the other end. The new value $a[j]$ must become the last value of these two sequences. In the first case we should delete all the values that are smaller than $a[j]$ and, if this value was not present in UP_{j-1}, augment the sequence by $a[j]$. If $a[j]$ was present in UP_{j-1}, then we need not augment the sequence with anything, but we must update the index of this value to j—this is its last occurrence now. Of course, we do the same with the other sequence. We delete all the greater values and add it at the end, or update the index if this value has been found there. Performing these actions will preserve the invariant of the loop: the sequences *UP* and *DOWN* are satisfying the definition, and the last occurrences of the values are memorized.

Now it is time to check whether the new element $a[j]$ has increased the longest t-stable segment so far. It is enough to check whether $j - i + 1$ is greater than the record.

Our algorithm now has three phases:

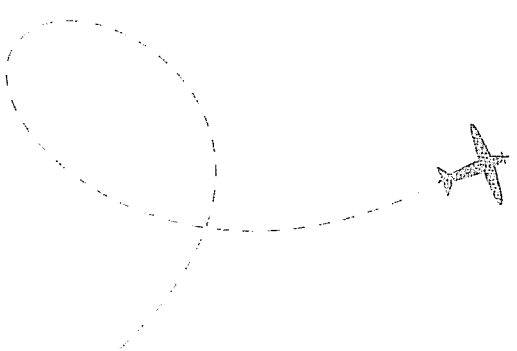
1. The updating of *UP* and *DOWN* from the beginning.
2. The updating of *UP* and *DOWN* from the end.
3. The updating of the record.

Both of our sequences can be maintained with the help of bi-queues, i.e. structures that allow the insertion and deletion of elements from both sides. The following operations will be needed (assume that the bi-queue elements will have two fields, *val* and *ind*, representing the value and the index of a segment element):

- Init(q)—to initialize an empty bi-queue q,
- Empty(q)—to verify the emptiness of q,
- First(q)—to return the value of the first element of q,
- DeleteFirst(q)—to delete the first element of q and return its value,
- Last(q)—to return the value of the last element of q,
- DeleteLast(q)—to delete the last element of q and return its value,
- InsertLast(q, x)—to insert x at the end of q.

Such a data structure can easily be implemented, for instance, with the help of a doubly-linked list or in an array representing a circular buffer. Each of these operations can be performed in constant time.

```
Init(UP)
Init(DOWN)
last_up := 0; last_down := 0
record := 1
for j := 1 to n do
    x.ind := j
    x.val := a[j]
    while (not Empty(UP)) and (First(UP).val − t > a[j]) do
        last_up := DeleteFirst(UP).ind
    while (not Empty(DOWN)) and (First(DOWN).val + t < a[j]) do
        last_down := DeleteFirst(DOWN).ind
    i := max(last_up + 1, last_down + 1)
    {End of phase 1}
    while (not Empty(UP)) and (Last(UP).val < a[j]) do
        DeleteLast(UP)
    if Empty(UP) or (Last(UP).val > a[j]) then
        InsertLast(UP, x)
    else
        Last(UP).ind := j
    while (not Empty(DOWN)) and (Last(DOWN).val > a[j]) do
        DeleteLast(DOWN)
    if Empty(DOWN) or (Last(DOWN).val < a[j]) then
        InsertLast(DOWN, x)
    else
        Last(DOWN).ind := j
    {End of phase 2}
    if j − i + 1 > record then
        record := j − i + 1
return record
```



Let's analyze the complexity of this solution. The first impression is that the complexity is not linear. Inside the outermost loop "for each j" we have two loops, each of them processing potentially linearly long sequences *UP* and *DOWN*. But the overall processing cannot take more than linear time, because the inner loops delete the elements, which had to be inserted previously. And at each loop iteration we can insert only one element for each j. So the total number of deletions is linear too.

Range Minimum Query Another idea that results in a linear solution is the general RMQ algorithm. This allows us to find the minimal and maximal values in a given segment of an array quickly. After preprocessing, which is linear with respect to the sequence length, the answers are given in constant time for each desired segment. However, this algorithm is a bit tricky to implement.

To sum up

This problem exposes a very typical case: we can improve the first idea we have by several orders of magnitude, but doing so requires quite a lot of effort and some knowledge. On this occasion we have achieved a tremendous decrease in complexity: from $O(n^3)$ (or even $O(n^4)$) to linear. Just as in real life, "cheap" does not usually mean "easy"; but it does pay off.