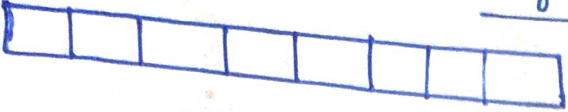


Input:  \rightarrow num's element can be +ve, -ve, or 0.

Output: longest ^{consecutive} subsequence length.

Sequence: A countable collection of objects in which repetition is allowed and order matters.

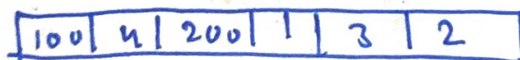
Sub-sequence: A sub sequence is a portion from sequence that can be formed by removing one or more elements from sequence without disturbing other elements' order.

For example: Set of natural numbers (N) is a sequence
And a function $5n+3$ represents a subsequence from that sequence

$$N \equiv \{1, 2, 3, 4, 5, \dots\}$$

$$f(n) = 5n+3 \equiv \{8, 13, 18, \dots\}$$

Example from question



\Downarrow can be seen as a consecutive subsequence

(1, 2, 3, 4)

4 \leftarrow max subsequence length.

Brute force

- 1) Sort the array ($O(n \log n)$)
- 2) As it is consecutive increase, length⁺⁺ counter when $\text{prevElement} + 1 == \text{next}$
and when $(\text{prevElement} + 1) \neq \text{next}$
- 3) Calculate ~~max~~ next length = 1;
by using max funcⁿ.
- 4) There are '2' corner cases to be handled here
 - 1) Equality check b/w prev and next elements
 - 2) ~~returning~~ '0' elements check \therefore output = 1
(as single element is always consecutive)
- 5) Return the ~~max~~ max output from length and max length
(This is recalculated because there can be cases when elements are more than one but repetitive for example [0,0])
where output should be '1'

Pseudocode

```
if (nums.size() == 0)
    return 0;
if (nums.size() == 1)
    return 1;
// Taking loop from i=1 because of ending comparison inside would have to end before end element
maxlength = 0;
length = 1;
for i = 1 to n
    if nums[i-1] + 1 == nums[i]
    if nums[i-1] != nums[i]
        if nums[i-1] + 1 == nums[i]
            length++; // current length
    else
        length = 1;
maxlength = max(maxlength, length);
return max(maxlength, length); // when both variables are placed globally do this
```

Day Run

1	2	3	4	5	7	8	9	10
0	1	2	3	4	5	6	7	8

i	length	max length	nums[i]	nums[i-1]
0	1	1	1	
1	2	2	2	1
2	3	3	3	2
3	4	4	4	3
4	1	5	5	4
5	2		6	6
6	3		7	7
7	4		8	8
8	5		9	9
9			10	10

max = 5

0	1	2	1
---	---	---	---

↓ sort

0	1	1	2
0	1	2	3

i	l	m	nums[i]	nums[i-1]
0	1	1		
1	2	2	1	1
2	2	2	1	1
3	3	3	2	2

[0, 0]

[0, 0]

i	l	m	nums[i]	nums[i-1]
0	1	1		
1	2	2	1	1

↑
coming from last return.

Optimised approach

- 1.) Store key : value \equiv element : boolean value in hashmap
 - a. \Rightarrow This is to confirm that you want to iterate over it while calculating longest ~~sub~~ consecutive subsequence.
(Not clear??, would be in next step)
- 2.) ~~Check~~ Traverse over map and invert to false to those whose previous elements exist.
Why??
Because if the previous element ~~doesn't~~ exists there is no need to traverse current element as it will ~~have no~~ consecutive elements have been traversed already in previous traversals.
- 3.) Increase the length ^{while} if (curr + length j) is found where 'j' is the position ^{of next ele} after each iteration if condition is true.
- 4.) check max length
- 5.) return max length

Pseudocode

Optimized.

```
for if nums.size() == 0  
    return 0;
```

```
if nums.size() == 1  
    return 1;
```

unordered_map <int, bool> mp; // to store elements' check
that needed to
be traversed

```
for (auto& it : nums)  
    mp[it] = true;
```

```
for i to n
```

```
    if mp.count(nums[i] - 1) > 0  
        mp[nums[i]] = false;
```

```
int max_length = 0;
```

```
for (int i = 0; i < nums.length
```

```
    for i to n
```

```
        if mp[nums[i]] == true
```

```
            length = 0
```

```
            while (mp.find(nums[i] + length)  
                    != mp.end())
```

```
                length++;
```

```
            max_length = max(length, max_length);
```

```
return max_length;
```

$[0, 1, 2, 1]$
0 1 2 3

↓ map

$\begin{bmatrix} 0 & - & T \\ 1 & - & T \\ 2 & - & T \end{bmatrix} \rightarrow \begin{bmatrix} 0 & - & T \\ 1 & - & F \\ 2 & - & F \end{bmatrix}$

∴ 0 → 1 → 2
length = 3
maxlength = 3