Integer Replacement

Given a positive integer n and you can do operations as follow:

1. If n is even, replace n with `n/2`.
2. If n is odd, you can replace n with either `n + 1` or `n - 1`.

What is the minimum number of replacements needed for n to become 1?

Example 1:

```
Input:
8

Output:
3

Explanation:
8 -> 4 -> 2 -> 1
```

Example 2:

```
Input:
7

Output:
4

Explanation:
7 -> 8 -> 4 -> 2 -> 1
or
7 -> 6 -> 3 -> 2 -> 1
```

Solution 1

I really think it should be tagged medium because there are many subtleties and good understanding of binary arithmetic is required.

The first step towards solution is to realize that you're allowed to remove the LSB only if it's zero. And to reach the target as fast as possible, removing digits is the best way to go. Hence, even numbers are better than odd. This is quite obvious.

What is not so obvious is what to do with odd numbers. One may think that you just need to remove as many 1's as possible to increase the evenness of the number. Wrong! Look at this example:

```
111011 -> 111010 -> 11101 -> 11100 -> 1110 -> 111 -> 1000 -> 100 -> 10 -> 1
```

And yet, this is not the best way because

```
111011 -> 111100 -> 11110 -> 1111 -> 10000 -> 1000 -> 100 -> 10 -> 1
```

See? Both `111011 -> 111010` and `111011 -> 111100` remove the same number of 1's, but the second way is better.

So, we just need to remove as many 1's as possible, doing +1 in case of a tie? Not quite. The infamous test with n=3 fails for that strategy because `11 -> 10 -> 1` is better than `11 -> 100 -> 10 -> 1`. Fortunately, that's the only exception (or at least I can't think of any other, and there are none in the tests).

So the logic is:

1. If `n` is even, halve it.
2. If `n=3` or `n-1` has less 1's than `n+1`, decrement `n`.
3. Otherwise, increment `n`.

Here is an example of such a solution in Java:

```java
public int integerReplacement(int n) {
    int c = 0;
    while (n != 1) {
        if ((n & 1) == 0) {
            n >>>= 1;
        } else if (n == 3 || Integer.bitCount(n + 1) > Integer.bitCount(n - 1)) {
            --n;
        } else {
            ++n;
        }
        ++c;
    }
    return c;
}
```

Of course, doing `bitCount` on every iteration is not the best way. It is enough to examine the last two digits to figure out whether incrementing or decrementing will give more 1's. Indeed, if a number ends with 01, then certainly decrementing is the way to go. Otherwise, if it ends with 11, then certainly incrementing is at least as good as incrementing (`*011 -> *010 / *100`) or even better (if there are three or more 1's). This leads to the following solution:

```java
public int integerReplacement(int n) {
    int c = 0;
    while (n != 1) {
        if ((n & 1) == 0) {
            n >>>= 1;
        } else if (n == 3 || ((n >>> 1) & 1) == 0) {
            --n;
        } else {
            ++n;
        }
        ++c;
    }
    return c;
}
```

An alternative approach to intuitive algorithm was very well put by@dettier in a discussion: you should create as many trailing zeroes as you can. This way you can avoid the tie-breaking trap (there can be no ties), but you'll still have to handle the n=3 exception separately.

written by SergeyTachenov original link here

Solution 2

When n is even, the operation is fixed. The only unknown procedure is when it is odd. When n is odd it can be written into the form n = 2k+1 (k is a non-negative integer.). That is, n+1 = 2k+2 and n-1 = 2k. Then, (n+1)/2 = k+1 and (n-1)/2 = k. So the one of (n+1)/2 and (n-1)/2 is even, another is odd. And the "best" case of this problem is to divide as much as possible. Because of that, always pick n+1 or n-1 based on if it can be divided by 4. The only special case of that is when n=3 you would like to pick n-1 rather than n+1.

```java
public int integerReplacement(int n) {
    if(n==Integer.MAX_VALUE) return 32; //n = 2^31-1;
    int count = 0;
    while(n>1){
        if(n%2==0) n/=2;
        else{
            if((n+1)%4==0&&(n-1!=2)) n+=1;
            else n-=1;
        }
        count++;
    }
    return count;
}
```

written by Nakanu original link here

Solution 3

For this problem, if we look at the binary form of each number, we can get the idea that for each '1' (except for the first '1') it counts to two steps, for each '0', it counts to one step.
So our goal is to use +1 or -1 to reduce steps.

For example,
13 = 1101
If we plus one, we can get 1110; if we reduce one, we can get 1100;
1110 needs 2+2+1 = 5 steps, while 1100 only needs 2+1+1 = 4 steps, so we choose n-1 in this step.

Use long to avoid overflow (if n is Integer.MAX_VALUE).

```
public class Solution {
    public int integerReplacement(int n) {
        long N = n;
        long small,big;
        int cnt = 0;
        while( N != 1){
         small = (N  & ( N -1));
         big = ( N & (N + 1));
         if( (N & 1) == 0){
          N >>= 1;
         }
         else if ( (small & (small-1)) <= (big & (big-1))){
          N = N - 1;
         }
         else{
          N = N +1;
         }
         cnt++;
        }
        return cnt;
    }
}
```

written by ZebraRabbit original link here