Rokos

Codelity - 2022 Aug12 *(3 questions in 1.5 hours)*

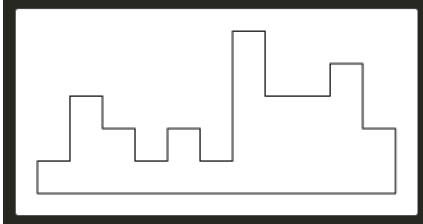
Question 1

Your room is being decorated. On the largest wall you would like to paint a skyline. The skyline consists of rectangular buildings arranged in a line. The buildings are all of the same width, but they may have different heights. The skyline shape is given as an array A whose elements specify the heights of consecutive buildings.

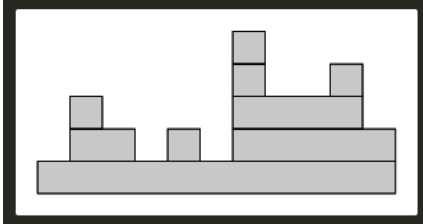
For example, consider array A such that:

```
A[0]  = 1
A[1]  = 3
A[2]  = 2
A[3]  = 1
A[4]  = 2
A[5]  = 1
A[6]  = 5
A[7]  = 3
A[8]  = 3
A[9]  = 4
A[10] = 2
```

The shape specified by this array is represented by the figure below.



You would like to paint the skyline using continuous horizontal brushstrokes. Every horizontal stroke is one unit high and arbitrarily wide. The goal is to calculate the minimum number of horizontal strokes needed. For example, the above shape can be painted using nine horizontal strokes.



Starting from the bottom, you can paint the skyline in horizontal stripes with 1, 3, 2, 2, 1 strokes per respective stripe.

Write a function:

```
int solution(vector<int> &A);
```

that, given a non-empty array A consisting of N integers, returns the minimum number of horizontal brushstrokes needed to paint the shape represented by the array.

The function should return −1 if the number of strokes exceeds 1,000,000,000.

For example, given array A as described above, the function should return 9, as explained above.

On the other hand, for the following array A:

```
A[0] = 5
A[1] = 8
```

the function should return 8, as you must paint one horizontal stroke at each height from 1 to 8.

For the following array:

```
A[0] = 1
A[1] = 1
A[2] = 1
A[3] = 1
```

the function should return 1, as you can paint this shape using a single horizontal stroke.

Write an **efficient** algorithm for the following assumptions:

- N is an integer within the range [1..100,000];
- each element of array A is an integer within the range [1..1,000,000,000].

My solution (it gives correct answer, but not optimum in terms of speed)

```
int num_stroke_at_height(const vector<int>& A, int height)
{
    bool stroke = false;
    int count = 0;

    if (A[0] >= height)
    {
        stroke = true;
        ++count;
    }
    for(int n=1; n!=A.size(); ++n)
    {
        if (!stroke)
        {
            if (A[n] >= height)
            {
                stroke = true;
                ++count;
            }
        }
        else
        {
            if (A[n] < height)
            {
                stroke = false;
            }
        }
    }
    return count;
}
```

```
int solution(vector<int> &A)
{
    int height = 1;
    int count = 0;
    while(true)
    {
        auto temp = num_stroke_at_height(A, height);

        if (temp == 0)
        {
            return count;
        }
        count += temp;
        if (count > 1e9)
        {
            return -1;
        }
        ++height;
    }
    return count;
}
```

This solution is not optimal, it is $O(N^2)$, I think it can be done in O(N) using stack-trick (as you can see from the figure in the question, it is like the call stack diagram in my C++ chapter 1). The algo is :

For each item in vector :

- if it is greater than the top of stack, push it
- if it is smaller than the top of stack, pop the stack until the new item is greater
- if it is the same as the top of stack, do nothing, i.e. don't push don't pop

sum of pop-difference in call stack
= sum of push-difference in call stack
= number of stroke needed
= number of function called / return in call stack

```
// This function is not verified.
std::uint32_t num_of_stroke(const std::vector<std::uint32_t>& input)
{
	std::stack s;
	std::uint32_t count = input[0];
	s.push(input[0]);

	for(std::uint32_t n=0; n!=input.size(); ++n)
	{
		if (A[n] > s.top())
		{
			count += A[n] - s.top();
			s.push(A[n]);
		}
		else
		{
			while(A[n]<s.top()) s.pop();
		}
	}
}

// You can also implement as pop-difference.
```

Question 2

You are given a string S of length N which encodes a non-negative number V in a binary form. Two types of operations may be performed on it to modify its value:

- if V is odd, subtract 1 from it;
- if V is even, divide it by 2.

These operations are performed until the value of V becomes 0.

For example, if string S = "011100", its value V initially is 28. The value of V would change as follows:

- V = 28, which is even: divide by 2 to obtain 14;
- V = 14, which is even: divide by 2 to obtain 7;
- V = 7, which is odd: subtract 1 to obtain 6;
- V = 6, which is even: divide by 2 to obtain 3;
- V = 3, which is odd: subtract 1 to obtain 2;
- V = 2, which is even: divide by 2 to obtain 1;
- V = 1, which is odd: subtract 1 to obtain 0.

Seven operations were required to reduce the value of V to 0.

Write a function:

```
int solution(string &S);
```

that, given a string S consisting of N characters containing a binary representation of the initial value V, returns the number of operations after which its value will become 0.

Examples:

1. Given S = "011100", the function should return 7. String S represents the number 28, which becomes 0 after seven operations, as explained above.

2. Given S = "111", the function should return 5. String S encodes the number V = 7. Its value will change over the following five operations:

- V = 7, which is odd: subtract 1 to obtain 6;
- V = 6, which is even: divide by 2 to obtain 3;
- V = 3, which is odd: subtract 1 to obtain 2;
- V = 2, which is even: divide by 2 to obtain 1;
- V = 1, which is odd: subtract 1 to obtain 0.

3. Given S = "1111010101111", the function should return 22.

4. Given string S consisting of "1" repeated 400,000 times, the function should return 799,999.

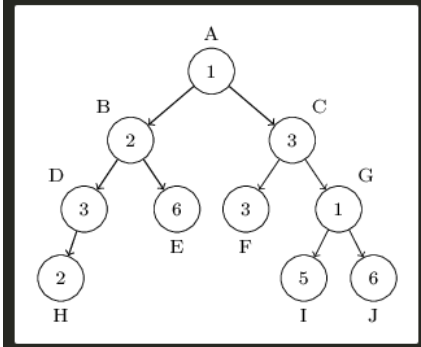
Write an **efficient** algorithm for the following assumptions:

- string S consists only of the characters "0" and/or "1";
- N, which is the length of string S, is an integer within the range [1..1,000,000];
- the binary representation is big-endian, i.e. the first character of string S corresponds to the most significant bit;
- the binary representation may contain leading zeros.

```
ans = input.size() - num_of_leading_zeros + num_of_ones - 1
```

Question 3

In this task we process binary trees containing integers in their nodes. An example tree is shown in the figure below:



Assume that the following declarations are given:

```
struct tree {
  int x;
  tree * l;
  tree * r;
};
```

This structure represents a tree, where:

- `x` is the integer contained in the node;
- `l` points to the left subtree or, if there is no left subtree, is equal to `NULL`;
- similarly, `r` points to the right subtree or is equal to `NULL`.

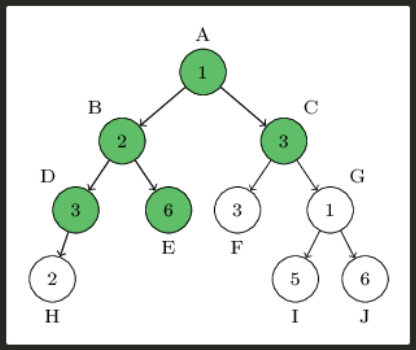
A *distinct path* is a sequence of nodes that starts at the root (the topmost node), follows the tree edges, and contains only distinct values.

For example:

- the sequence [A, B, D] is a distinct path
- [A, B, J] is not a distinct path, because it doesn't follow the tree edges
- [C, G, J] is not a distinct path, because it doesn't start at the root
- [A, C, F] is not a distinct path, because it contains the value 3 twice

We would like to find the number of nodes on the longest distinct path.

In the example tree, there are two distinct paths of maximum size: [A, B, D] and [A, B, E]. They consist of three nodes each, so the answer should be 3.



Write a function:

```
int solution(tree * T);
```

that, given a binary tree T consisting of N nodes, returns the number of nodes on the longest distinct path that starts at the root.

Write an **efficient** algorithm for the following assumptions:

- N is an integer within the range [1..50,000];
- the height of tree T (number of edges on the longest path from root to leaf) is within the range [0..3,500];
- each value in tree T is an integer within the range [1..N].

My solution

```
#include <queue>
#include <unordered_set>

struct item
{
    tree* node;
    std::unordered_set<int> path;
};

int solution(tree* root)
{
    queue<item> q;
    if (root)
    {
        q.push(item{root, {}});
    }
    else return 0;

    std::size_t max_count = 0;
    while(!q.empty())
    {
        auto& this_node = q.front().node;
        auto& this_path = q.front().path;
        if (this_path.find(this_node->x)==this_path.end())
        {
            this_path.insert(this_node->x);
            max_count = std::max(max_count, this_path.size());

            if (this_node->l)
            {
                q.push(item{this_node->l, this_path});
            }
            if (this_node->r)
            {
                q.push(item{this_node->r, this_path});
            }
        }
        q.pop();
    }
    return (int)(max_count);
}

```