LeetCode 0

String to Integer (atoi) 4.58 (163 votes)

String to Integer

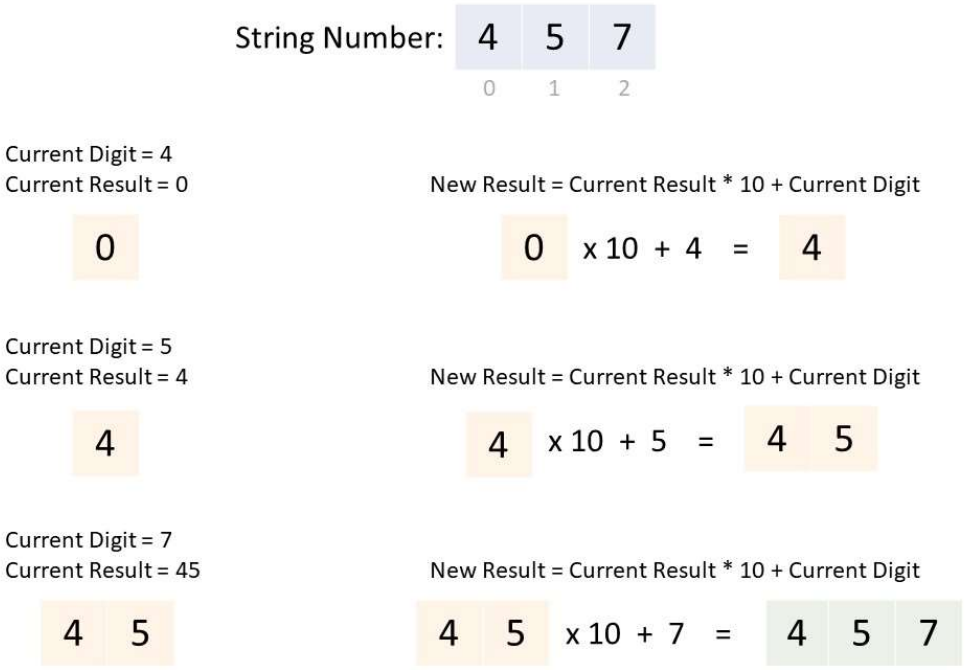
LeetCode Admin Dec 08, 2021

Solution

Overview

We need to implement a function that converts the given string into a signed 32-bit integer. Intuitively, we could build the output number out of the input string by iterating over it character by character. However, we stop building the number when a non-digit character is spotted, or the number becomes too large to fit inside a 32-bit signed integer. In the latter case, we need to clamp the result to fit the range.

We will build the integer one character at a time. As we traverse the string from left to right, for each digit character, we will shift all digits in the current integer to the left by one place (this is done by multiplying the integer by 10). Then, we can simply add the current digit to the unit place of the integer. To better understand how this process works, let's look at an example:



The key to solving this problem is carefully reading the problem statement, following the rules given, thinking about edge cases, and keeping your code simple.

> **Interview Tip:** Asked a question like this in an interview? Be sure to communicate thoroughly with your interviewer to make sure you're covering all cases. In this problem, the rules are very thorough because there is no interviewer to communicate with. However, in an interview, each of these rules is a potential question to ask the interviewer if the rule is not already stated.

Approach 1: Follow the Rules

Intuition

Given the rules outlined by the problem's description, we can iterate over the input string and use the given rules to validate it.

First read through the problem statement **very carefully**. Let's see what are all the possible characters in the input string:

- Whitespaces **(' ')**
- Digits **('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')**
- A sign **('+' or '-')**
- Anything else (alphabetic characters, symbols, special characters, etc.)

And write down all the rules for building the integer for each one these characters which will help us in writing down the conditions while building the algorithm.

Rules

- Whitespaces:
  - If any whitespaces occur at the beginning of the input string, we discard them.
  - However, if whitespace occurs anywhere else in the input, then we stop and discard the rest of the input.

```
'   1234' => 1234 (whitespaces at beginning are removed)
'      4' => 4    (whitespaces at beginning are removed)
' 12   4' => 12   (only the leading whitespaces are removed)
```

- Digits:
  - Discard any leading zeros.
  - Read in all the digit characters until the first non-digit character or the end of the input occur and append those to the output number.
  - If no digits were found, return 0.

```
'12345 567 v' => 12345 (digits are appended until a non-digit character occurs)
'00123'       => 00123 => 123 (0s in the beginning of the numbers are discarded)
```

- Sign:
  - There could be at most one sign character presented at the beginning, or after skipping some whitespaces from the beginning of the input string. Otherwise, a sign anywhere else in the input string is not valid and is considered a non-digit character and we stop building the integer.
  - If a **'+'** or no sign is present, the final number will be a positive integer. On the other hand, the final number will be negative if **'-'** is the first non-whitespace character in the string.

```
'123'  => 123 (a number with no sign is a positive number)
'+123' => 123 (a number with '+' sign is a positive number)
'-12'  => -12 (a number with '-' sign is a negative number)
'-+12' => 0   (another sign after one sign is considered as non-digit character)
```

- Anything else:
  - If any other character not covered by previously defined rules is spotted, we stop building the output number.

```
'-23a45 567 v' => -23 (we stopped when 'a' character occured)
'123 45 567 v' => 123 (we stopped when a space character occured)
'a+123 bcd 45' => 0   (we stopped when 'a' character occured in the beginning)
```

- 32-bit Integer Range:
  - If the integer exceeds $2^{31} - 1$ then it will be clamped to $2^{31} - 1$.
  - And if the integer becomes less than -2^{31} then it will be clamped to -2^{31}.

```
'12345' => 12345 (integer is in 32-bit range)
'9999999999999' => 2^31-1 (integer overflow)
'-999999999999' => -2^31   (integer underflow)
```

How to check overflows/underflows?

If we were using a `long`, `BigInteger`, or any other numeric data types to store the integers, we could check if the integer exceeds the `32-bit` range, stop building the output number, and return the clamped value.

```
num = num * 10 + digit
if num > 2^31 - 1 then return 2^31 - 1
else if num < -2^31 then return -2^31
```

However, here we will assume our environment doesn't allow us to use these data types which could be a constraint imposed by the interviewer. But we can't directly use a `32-bit` integer to store the final result.

For example, assume currently `result` is `1000000000` and `digit` is `1`, we can't append the current digit to result as `10000000001` is more than $2^{31} - 1$. So, performing the `result = result * 10 + digit` operation will result in **Runtime Error**.

Hence, first we need to check if appending the digit to the result is safe or not. If it is safe to append then update the result. Otherwise, handle the overflow/underflow.

Let's first consider the case for overflow.

We will denote the maximum 32-bit integer value $2^{31} - 1 = 2147483647$ with **INT_MAX**, and we will append the digits one by one to the final number.

So there could be 3 cases:

- Case 1: If the current number is less than **INT_MAX / 10 = 214748364**, we can append any digit, and the new number will always be less than **INT_MAX**.

```
'214748363' (less than INT_MAX / 10) + '0' = '2147483630' (less than INT_MAX)
'214748363' (less than INT_MAX / 10) + '9' = '2147483639' (less than INT_MAX)
'1' (less than INT_MAX / 10) + '9' = '19' (less than INT_MAX)
```

- Case 2: If the current number is more than **INT_MAX / 10 = 214748364**, appending any digit will result in a number greater than **INT_MAX**.

```
'214748365' + '0' = '2147483650' (more than INT_MAX)
'214748365' + '9' = '2147483659' (more than INT_MAX)
'2147483646' + '8' = '21474836468' (more than INT_MAX)
```

- Case 3: If the current number is equal to **INT_MAX / 10 = 214748364**, we can only append digits from **0-7** such that the new number will always be less than or equal to **INT_MAX**.

```
'214748364' + '0' = '2147483640' (which is less than INT_MAX)
'214748364' + '7' = '2147483647' (which is equal to INT_MAX)
'214748364' + '8' = '2147483648' (which is more than INT_MAX)
```

Similarly for underflow.
The minimum 32-bit integer value is $-2^{31} = -2147483648$ denote it with **INT_MIN**.

We append the digits one by one to the final number.
Just like before, there could be 3 cases:

- Case 1: If the current number is greater than **INT_MIN / 10 = -214748364**, then we can append any digit and the new number will always be greater than **INT_MIN**.

- Case 2: If the current number is less than **INT_MIN / 10 = -214748364**, appending any digit will result in a number less than **INT_MIN**.

- Case 3: If the current number is equal to **INT_MIN / 10 = -214748364**, then we can only append digits from **0-8**, such that the new number will always be greater than or equal to **INT_MIN**.

Notice that **cases 1 and 2** are similar for **overflow** and **underflow**. The only difference is **case 3**: for overflow, we can append any digit between **0 and 7**, but for underflow, we can append any digit between **0 and 8**.

So we can combine both the underflow and overflow checks as follows:

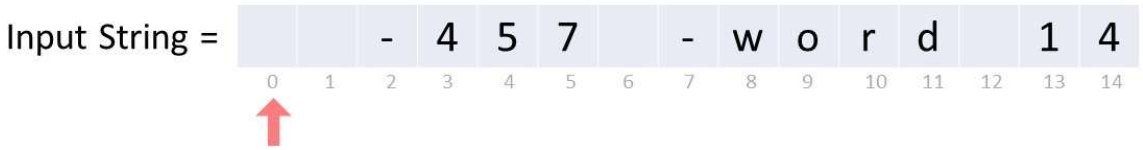
- Initially, store the sign for the final result and consider only the absolute values to build the integer and return the final result with a correct sign at the end.

- If the current number is less than **214748364 = (INT_MAX / 10)**, we can append the next digit.
- If the current number is greater than **214748364**:
  - And, the sign for the result is **'+'**, then the result will **overflow**, so return **INT_MAX**;
  - Or, the sign for the result is **'-'**, then the result will **underflow**, so return **INT_MIN**.
- If the current number is equal to **214748364**:
  - If the next digit is between **0-7**, the result will always be in range.
  - If, next digit is **8**:
    - and the sign is **'+'** the result will **overflow**, so return **INT_MAX**;
    - if the sign is **'-'**, the result will not **underflow** but will still be equal to INT_MIN, so that we can return **INT_MIN**.
  - But if, the next digit is greater than **8**:
    - and the sign is **'+'** the result will **overflow**, so return **INT_MAX**;
    - if the sign is **'-'**, the result will **underflow**, so return **INT_MIN**.

> **Note:** We do not need to handle **0-7** for positive and **0-8** for negative integers separately. If the sign is **negative** and the current number is **214748364**, then appending the digit **8**, which is more than **7**, will also lead to the same result, i.e., **INT_MIN.**

Algorithm

1. Initialize two variables:
   - `sign` (to store the sign of the final result) as `1`.
   - `result` (to store the 32-bit integer result) as `0`.
2. Skip all leading whitespaces in the input string.
3. Check if the current character is a `'+'` or `'-'` sign:
   - If there is no symbol or the current character is `'+'`, keep `sign` equal to `1`.
   - Otherwise, if the current character is `'-'`, change `sign` to `-1`.
4. Iterate over the characters in the string as long as the current character represents a digit or until we reach the end of the input string.
   - Before appending the currently selected digit, check if the 32-bit signed integer range is violated. If it is violated, then return `INT_MAX` or `INT_MIN` as appropriate.
   - Otherwise, if appending the digit does not result in overflow/underflow, append the current digit to the `result`.
5. Return the final `result` with its respective sign, `sign * result`.



Sign = 1 (final result will be positive)
Result = 0
Current Character ` (space character)

All the space characters in the beginning of the input string will be skipped.

1 / 7

Implementation

C++ | Java | JavaScript | Python3 — Copy

```
class Solution {
public:
    int myAtoi(string input) {
        int sign = 1;
        int result = 0;
        int index = 0;
        int n = input.size();

        // Discard all spaces from the beginning of the input string.
        while (index < n && input[index] == ' ') {
            index++;
        }

        // sign = +1, if it's positive number, otherwise sign = -1.
        if (index < n && input[index] == '+') {
            sign = 1;
            index++;
        } else if (index < n && input[index] == '-') {
            sign = -1;
            index++;
        }

        // Traverse next digits of input and stop if it is not a digit.
        // End of string is also non-digit character.
        while (index < n && isdigit(input[index])) {
            int digit = input[index] - '0';
```

Complexity Analysis

If N is the number of characters in the input string.

- Time complexity: $O(N)$

  We visit each character in the input at most once and for each character we spend a constant amount of time.

- Space complexity: $O(1)$

  We have used only constant space to store the sign and the result.

Approach 2: Deterministic Finite Automaton (DFA)

Intuition

While the previous approach would likely be sufficient for an interview, the approach is specific to this problem. Here we will present an approach that uses DFA which is a more generalized approach that can also be applied to similar problems that would otherwise require writing many nested if else conditions which could become very complex.

The Deterministic Finite Automaton approach may feel familiar to you if you have previously studied TOC (Theory Of Computation). If you're unfamiliar with DFA, we will provide a short introduction below, but we encourage you to read more about DFA outside of this article as well.

▶ Here's a short introduction to DFA. (click to expand)

▶ Some other LeetCode problems which can be solved using DFA:

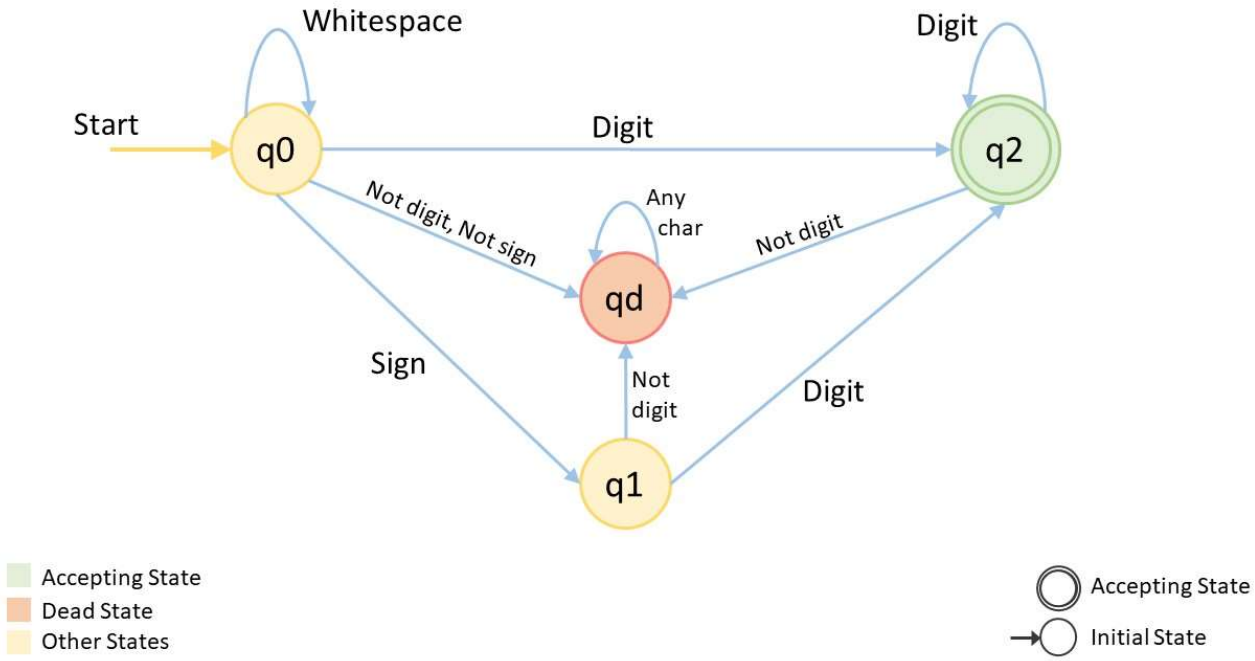
Now that we have some basic knowledge about state machines, let's try to approach this problem by using a state machine. We can develop a simple state machine where we give the input string characters one by one as the input to the machine and it will produce the desired integer as output.

We can say, initially we are in some starting state and each time we read a character in the input string, we either stay in the current state or transition to a new state. If at any step the state becomes invalid (i.e. when a non-digit character is spotted, or the 32-bit signed integer range is reached) then we can stop building the integer.

What we've described above is a lot like a deterministic finite automaton as in DFAs there is only one path for specific input from the current state to the next state.

The first step is to design our DFA. Picture the DFA as a directed graph, where each node is a state, and each edge is a transition labeled with a character.

Naturally, DFA could be represented as a set of if-else conditions. The following diagram presents a common way to write these if-else conditions.

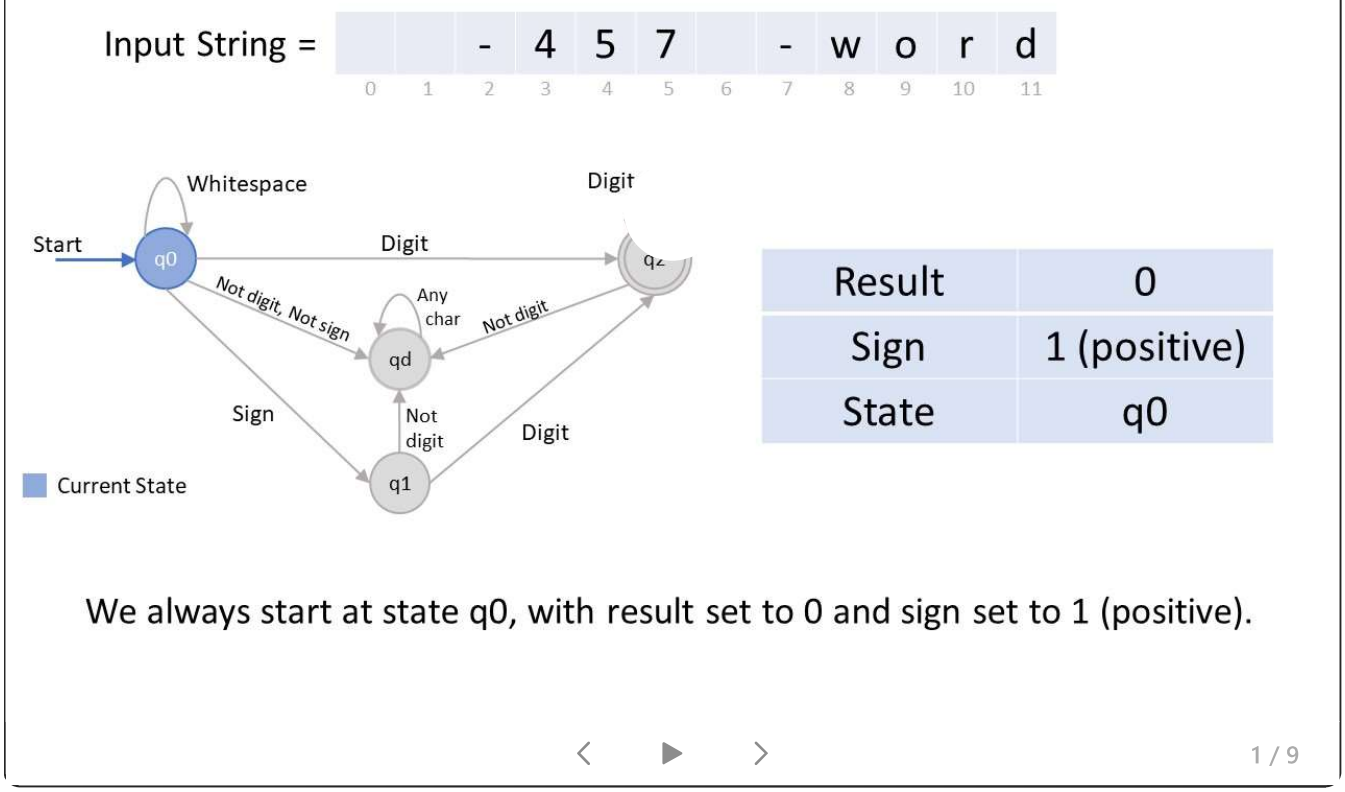


1. **State** q_0: represents the initial state at the beginning of the input string.
    - Discard any leading whitespace characters in the beginning as per the rules given and remain in the same q_0 state for now.
    - If a sign character occurs, transition to state q_1.
    - If a digit character occurs, transition to state q_2.
    - Once a non-digit character is spotted, stop building the output number and immediately transition to a dead state q_d.
2. **State** q_1: we arrive at this state after the first sign character has been found.
    - After one sign character if now a digit occurs then we transition to state q_2.
    - Once a non-digit character is spotted, stop building the output number and immediately transition to a dead state q_d.
3. **State** q_2: we arrive at this state whenever the previous character was a digit.
    - Remain in the current state if the next character happened to be a digit character.
    - Once a non-digit character is spotted, stop building the output number and immediately transition to a dead state q_d.

4. **State** q_d: a dead state meaning one or more rules defined in the beginning have been violated. This state marks the end of the number-building process.

Algorithm

1. Initialize three variables:
   - `currentState` (to represent current state) as `q0`
   - `result` (to store result till now) as `0`
   - `sign` (to represent the sign of the final result) as `1`
2. For each character of the input string, if the current state is not `qd`:
   - If the current state is `q0`:
     - Stay in the same state if whitespaces occur.
     - If a sign occurs, change the sign variable to `-1` if it's a negative sign and change the state to `q1`.
     - If a digit occurs, append the current digit to the resulting number (clamp result if needed) and change the state to `q2`.
     - If anything else occurs, then stop building the number and transition to state `qd`.
   - If the current state is `q1`:
     - If a digit occurs, append the current digit to the resulting number (clamp result if needed) and change the state to `q2`.
     - If anything else occurs, stop building the result number and transition to state `qd`.
   - If the current state is `q2`:
     - If a digit occurs, append the current digit to the resulting number (clamp result if needed) and stay in the current state.
     - Anything else after a digit character will not be valid; hence, stop building the number and transition to state `qd`.
3. Return the final result with the respective sign, `result * sign`.



Implementation

C++ | Java | JavaScript | Python3 — Copy

```cpp
enum State { q0, q1, q2, qd };

class StateMachine {
    // Store current state value.
    State currentState;
    // Store result formed and it's sign.
    int result, sign;

    // Transition to state q1.
    void toStateQ1(char& ch) {
        sign = (ch == '-') ? -1 : 1;
        currentState = q1;
    }

    // Transition to state q2.
    void toStateQ2(int digit) {
        currentState = q2;
        appendDigit(digit);
    }

    // Transition to dead state qd.
    void toStateQd() {
        currentState = qd;
    }

    // Append digit to result, if out of range return clamped value.
    void appendDigit(int& digit) {
```

Complexity Analysis

If N is the number of characters in the input string.

- Time complexity: $O(N)$

  We iterate over the input string exactly once, and each state transition only requires constant time.

- Space complexity: $O(1)$

  We have used only constant space to store the state, sign, and result.

Share

Comments (131) Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

Article Commenting Rules

1. This comment section is for questions and comments regarding this **LeetCode article**. All posts must respect our LeetCode Community Rules.

2. Concerns about errors or bugs in the article, problem description, or test cases should be posted on LeetCode Feedback, so that our team can address them.

peterx123 Jan 13, 2022

This is question is the edge king.

360 Show 8 Replies Reply

kryuki Jan 13, 2022

What a painful problem this is!!! I really hate this...

132 Show 2 Replies Reply

nitinyadav3... Dec 10, 2021

Amazing article!! All leetcode solutions must have article like this!

98 Show 2 Replies Reply

CodingNoo... Feb 10, 2022

Is this written by an Assembly level coder>>>

40 Reply

ahmedtremo Jan 07, 2022

Damn, this problem took so long. It has a lot of test cases

57 Show 1 Replies Reply

introvert_0... Jan 14, 2022

First of all looking at the problem i thought it is an cake walk to solve the problem. But when i submitted my solution i am getting wrong answer. Everytime I see the test case which give me wrong answer I found a bug in my code. I submitted 5 wrong answers to question thinking that after that every wrong answer the code will work. At last 6th time I got accepted solution..
Never Ever write code without carefully think on it...

33 Show 1 Replies Reply

nicolattu

Jan 13, 2022

lol now I get it why 16% acceptance. lots of fun :D

43 Reply

Symbolistic

Jan 31, 2022

This is literally one hell of a stupid problem, most likely the definition of a stupid problem.
If we ever have to do this on the job, we have a bigger problem on our hands.

33 Show 1 Replies Reply

sudiptob2

Jan 14, 2022

can not hate this problem only because of this article (-_-)

21 Reply

ezbuckeye

Mar 18, 2022

a TRICK about one of the two cases that the result would over/underflow in approach 1:

```
(res==Integer.MAX_VALUE/10 && digit>Integer.MAX_VALUE%10)
```

if it is positive, it will overflow and return the Integer.MAX_VALUE.
If it is negative, it will either underflow or be the Integer.MIN_VALUE, so no matter it is which case, we can just return the Integer.MIN_VALUE.

6 Reply