myAtoi:

we need to implement a functon that converts the given string into a signed 31-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 signed integer. In the latter case, we need to clap the result to fit the range.

We will build the integer one characyer at a time. As we traverse the string from left to right, for each digit characyer, 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:

Graphical user interface

Description automatically generatedStr

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

Interview tip: Asked a question like this is an interview? Be sure to communicate throughly with your interviewee 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 questions to ask the interviewer if the rule is not already stated.

**Approach 1: Follow the rules**

**Intuition**

Given the rules outlines 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:

* Whitespace(‘’)
* Digits(“0”, “1”, 2, 3, 4, 5, 6, 7, 8, 9), in the form of string
* A sign (“+” or “-“)

And write down all the rules for building the integer for each one these characyers which will help us in writing down the condistions while building the algo.

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

‘12345 567’ => 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 neagtive if ‘-‘ is the first non-whitespace character in the string.

‘123’ => 123 ( anumber 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 characyer)

**Anything else:**

* If any other characyer not covered by previously defined rules is spotted, we stop building the output number.

‘-23a45 567 v’ => -23 (we stop when ‘a’ character occurred)

‘123 45 567 v’ => 123 (we stop when a space character occurred)

‘a+123 bcd 45’ => 0 (we stopped when ‘a’ character occurred 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

**How to check overflows/underflows?**

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 eill denote the maxinum 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
* Case 3: If the current number is more than INT\_MAX/10 = 21474748364, appending any digit will result in a number greater than INT\_MAX
  + ‘214748365’ + ‘0’ = ‘214748365’ (more than INT\_MAX)
  + ‘214748365’ + ‘9’ = ‘2147483659’ (more than INT\_MAX)
* Case 3: If the current 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 euqal to INT\_MAX

**Similar for underflow:**

The mininum 32-bit integer values 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 rgeater than INT\_MIN/10 = -214748364, then we can append any digit and the new number will always be greater than INT\_MIN.

Notice that cases 1 and 2 are similar for overflow and underflow.

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 the 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 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.
* Algorithm:
* 1: Initialize 2 variables:
  + Sign (to store the sign of the final result) as 1
  + Result (to sore 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 characyer 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 overfolow/underflow, append the current digit to the result
* 5: Return the final result with its respective sign, sign \* result

The skeleton of the algo is a loop that iterate through each cell in the grid. For each cell, we invoke the backtracking fynctuon (i.e backtrack()) to check if we would obtain a solution, starting from this very cell.

For the backtracking function backtracking(row, col, suffix), as a DFS algotithm, it is ofeten implemented as a recursive function. The function can be broke down into the following four steps:

Step 1: at the beginning, first we check if we reach the bottome case of the recursion, where the word to be matched is empty. i.e we have already found the match for each prefix of the word

Step 2: We then check If the current state is invalid,

+ either the position of the cell is out of the boundary of the board

+ or the letter in the current cell does not match with the first letter of the word

Step 3: If the current step is valid, we then start the exploration of backtracking with the strategy of DFS. First, we mark the current cell as visited, e.g. any non alphabetic letter will do. Then we iterate through the four possible directions, u

Step 4: At the end of the exploration, we revert the cell back to its original state. Finally we return the result of the exploration.