

**STRING**

# Problem – 3. Longest Substring Without Repeating Characters

Medium



LeetCode

[leetcode.com/problems/longest-substring-without-repeating-characters](https://leetcode.com/problems/longest-substring-without-repeating-characters)

## Problem Statement

- You are given a string and the goal is to find the longest substring without repeating characters

- **Example**

**Input:** "abcbd"

**Output:** 4 (abcd since "b" is repeated)

# Solution – 3. Longest Substring Without Repeating Characters

Medium



LeetCode

[leetcode.com/problems/longest-substring-without-repeating-characters](https://leetcode.com/problems/longest-substring-without-repeating-characters)

## Solution

- Use sliding window algorithm (left and right)
- Loop through the string
- Try to find if the current character is already added by using unordered set or bitmap
- If added, remove from the set alongside with others using left pointer
- If not, add to the unordered set or bitmap
- Maximum length will be  $\text{right} - \text{left} + 1$

# Example – 3. Longest Substring Without Repeating Characters

Medium



LeetCode

[leetcode.com/problems/longest-substring-without-repeating-characters](https://leetcode.com/problems/longest-substring-without-repeating-characters)

## Example

- String: abcbd. Our goal is to return 3 (**abc**bd)
- Initialize **maxLength = 0**
- Loop through the string

**Iteration 1:** left = 0, right = 0, string[left] = 'a',

bitmap = ['a'] ('a' is not in bitmap, add), **maxLength = max(maxLength, right - left + 1) = 1**

**Iteration 2:** left = 0, right = 1, string[right] = 'b'

bitmap = ['a','b'], **maxLength = 2**

**Iteration 3:** left = 0, right = 2, string[right] = 'c'

bitmap = ['a','b','c'], **maxLength = 3**

**Iteration 4:** left = 0, right = 3, string[right] = 'b'

bitmap = ['a','b','c','b']

'b' is already in the bitmap. start "clearing" the character using left:

**Iteration 4a:** left = 0, string[left] = 'a' is different from 'b', so remove 'a'

bitmap = ['b','c','b']

**Iteration 4b:** left = 1, string[left] = 'b' is the same as the repeated one, remove

bitmap = ['c','b']

**Iteration 5:** left = 1, right = 4, string[right] = 'd'

bitmap = ['c','b','d']

# Code – 3. Longest Substring Without Repeating Characters

Medium

## Code (unordered\_set)

- Use unordered\_set when question requires unicode chars

```
int lengthOfLongestSubstring(string s) {  
    int maxLength = 0;  
    int left = 0, right = 0;  
    // track the seen characters  
    unordered_set<char> seen;  
    for (right = 0; right < s.size(); ++right) {  
        char currentChar = s[right];  
        // if currentChar is in the set, clean  
        // the character and everything from left of it  
        // basically, reset the longest substring  
        while (seen.count(currentChar)) {  
            char c = s[left];  
            seen.erase(c);  
            left++;  
        }  
        // insert the current read character  
        seen.insert(currentChar);  
        // set max length  
        maxLength = max(maxLength, right - left + 1);  
    }  
    return maxLength;  
}
```

# Code – 3. Longest Substring Without Repeating Characters

Medium

## Code (bitmap)

- Using bitset: create a bitmask with 128 bits where each bit represent a character
- Optimal solution for ASCII since ASCII size is 127 characters
- Unicode / UTF-8 can represent over 1.1 million characters, so use **unordered\_set** approach instead

```
int lengthOfLongestSubstring(string s) {
    std::bitset<128> bitmask;
    uint32_t left = 0;
    uint32_t maxLength = 0;

    for (uint32_t right = 0; right < s.length(); ++right) {
        uint32_t bitIndex = s[right];
        // if char is already in the bitmask, move left until we reset the bits
        while (bitmask.test(bitIndex)) {
            bitmask.reset(s[left]);
            ++left;
        }

        bitmask.set(bitIndex);
        maxLength = std::max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

# Problem – 424. Longest Repeating Character Replacement

Medium

 [leetcode.com/problems/longest-repeating-character-replacement](https://leetcode.com/problems/longest-repeating-character-replacement)

## Problem

- You are given a **string s** and an **integer k**
- You can replace one character by any other uppercase English character **k** times
- Return the longest substring with the same character

- **Example:**

**Input:**

$s = \text{"ABAB"}, k = 2$

**Output:** 4

Replace the two 'A's with two 'B's or vice versa.

# Problem – 424. Longest Repeating Character Replacement

Medium

 [leetcode.com/problems/longest-repeating-character-replacement](https://leetcode.com/problems/longest-repeating-character-replacement)

## Solution

- Start with two pointers: left and right
- Keep track of the frequencies of each letter in a **vector<int>** since we know there are 26 characters
- Initialize **maxFreq** to keep track of the letter with maximum frequency
- Initialize **maxLength** to keep track of the maximum substring
- Go over the string, and for each iteration:
  - calculate the windowSize
  - calculate the maximum frequency
  - check how many replacements is needed. That is,  $\text{windowSize} - \text{maxFreq}$
  - if no replace can be done ( $k < \text{replaces}$ ) then move left pointer to the right



# Problem – 424. Longest Repeating Character Replacement

Medium

 [leetcode.com/problems/longest-repeating-character-replacement](https://leetcode.com/problems/longest-repeating-character-replacement)

**Code** Time:  $O(n)$  Space:  $O(1)$

```
int characterReplacement(string s, int k) {
    int left = 0;
    int maxLength = 0;
    int maxFreq = 0;
    vector<int> freq(26, 0);
    for (int right = 0; right < s.size(); ++right) {
        int index = s[right] - 'A';
        int windowSize = right - left + 1;
        // keep track of the frequencies
        freq[index]++;
        maxFreq = max(maxFreq, freq[index]);

        // check if the subwindow need to change
        int needReplace = windowSize - maxFreq;
        if (k < needReplace) {
            // need to move sub window
            int leftIndex = s[left] - 'A';
            freq[leftIndex]--;
            left++;
            windowSize = right - left + 1;
        }
        maxLength = max(maxLength, windowSize);
    }
    return maxLength;
}
```

# Problem – 76. Minimum Window Substring

Hard



LeetCode

[leetcode.com/problems/minimum-window-substring](https://leetcode.com/problems/minimum-window-substring)

## Problem

■ ...

# Solution – 76. Maximum Window Substring

Hard



LeetCode

[leetcode.com/problems/minimum-window-substring](https://leetcode.com/problems/minimum-window-substring)

## Solution

- **Grow** → first valid window: move right until the window has every required char (use a need table and a have table plus **formed == distinctNeeded** to know this)
- **Prune** ← from left: while the window is still valid, drop **s[left]** and advance left-stop as soon as removing a char would break validity
- **Record** current window length as a candidate answer.
- **Resume** growing right, repeating the **grow → prune ← record** cycle until right reaches the end.
- Two pointers only move forward

# Code – 76. Maximum Window Substring

Hard



LeetCode

[leetcode.com/problems/minimum-window-substring](https://leetcode.com/problems/minimum-window-substring)

**Code** Time:  $O(|s| + |t|)$  Space:  $O(k)$  where  $|s|$  means the size of "s" and  $|t|$  the size of "t".  $k$  is the number of distinct characters in  $t$

```
string minWindow(string s, string t) {
    if (t.size() > s.size()) return "";

    // characters I need (t)
    unordered_map<char, int> need;
    // current window
    unordered_map<char, int> window;
    int left = 0;
    int right = 0;
    int start = 0;

    // number of valid characters
    int valid = 0;
    int minLength = INT_MAX;

    // populate need
    // need['A'] = 1
    // need['B'] = 1
    // need['C'] = 1
    for (const auto& c : t) {
        need[c]++;
    }

    // traverse the string
    while (right < s.size()) {
        // current char
        char c = s[right];
        // increase right
        right++;

        // do we need this character?
        if (need.count(c)) {
            // add to the current window
            window[c]++;
            // have we reached the number of characters we need?
            // then increase valid. It doesn't matter if have more,
            // what matters is exactly the number
            if (window[c] == need[c]) {
                valid++;
            }
        }
    }
}
```

# Code – 76. Maximum Window Substring

Hard



LeetCode

[leetcode.com/problems/minimum-window-substring](https://leetcode.com/problems/minimum-window-substring)

**Code (continue)** Time:  $O(|s| + |t|)$  Space:  $O(k)$  where  $|s|$  means the size of "s" and  $|t|$  the size of "t".  $k$  is the number of distinct characters in  $k$

```
// this will run once our window is now valid,
// meaning having all characters from need
// now we want to prune this because we want the minimum
// window substring
while (valid == need.size()) {
    int windowSize = right - left;
    // minLength hold the global minimum substring
    // current valid windowSize is smaller, update it
    if (windowSize < minLength) {
        minLength = windowSize;
        // we need to keep track where the substring starts
        start = left;
    }

    // prune substring
    // check if s[left] is needed
    // is the character I'm pruning, needed?
    char charToPrune = s[left];
    left++;
    if (need.count(charToPrune)) {
        // ok we need this character, and the amount we have is
        // exactly what we need (we don't have more to 'spare')
        if (window[charToPrune] == need[charToPrune]) {
            // invalidate. So break the while loop and
            // continue moving right
            valid--;
        }
        // character is removed
        window[charToPrune]--;
    }
}
}
```

```
// return
if (minLength == INT_MAX) return "";
return s.substr(start, minLength);
}
```

# Problem – 242. Valid Anagram

Easy



LeetCode

[leetcode.com/problems/valid-anagram](https://leetcode.com/problems/valid-anagram)

## Problem

- You are given two strings **s** and **t**
- Return true if **t** is an anagram of **s**

- **Example:**

t = word

s = dwor

**Output:** true

both have the same number of same characters

# Problem – 242. Valid Anagram

Easy



LeetCode

[leetcode.com/problems/valid-anagram](https://leetcode.com/problems/valid-anagram)

## Solution

- Initialize a vector of integers to keep track of the count of each letter
- Loop over **s** and increase the count of each character found
- Then, loop over **t** and decrease the count of each character found
- Finally, loop over the vector and if there is one count greater than 0, return false

# Problem – 242. Valid Anagram

Easy



LeetCode

[leetcode.com/problems/valid-anagram](https://leetcode.com/problems/valid-anagram)

**Code** Time:  **$O(n)$**  Space:  **$O(1)$**

```
bool isAnagram(string s, string t) {  
    // count the number of characters in 's', store in a vector  
    // go over the vector and check if it's empty  
    vector<int> letters(26);  
    for (const auto& c : s) {  
        letters[c - 'a']++;  
    }  
    for (const auto& c : t) {  
        letters[c - 'a']--;  
    }  
    for (const auto& c : letters) {  
        if (c != 0) return false;  
    }  
    return true;  
}
```



# Problem – 49. Group Anagrams

Medium



LeetCode

[leetcode.com/problems/group-anagrams](https://leetcode.com/problems/group-anagrams)

## Problem

- You are given an array of strings, Example:  
`strs = ["eat", "tea", "tan", "ate", "nat", "bat"]`
- Group the anagrams together:  
`[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`
- No anagram of "bat", where "nat" and "tan" are anagram so they're grouped together

# Solution – 49. Group Anagrams

Medium



LeetCode

[leetcode.com/problems/group-anagrams](https://leetcode.com/problems/group-anagrams)

## Solution

- Go over each word
- Sort the words

**Example:** ["eat", "tea", "tan", "ate", "nat", "bat"]

**After sorting:** ["aet", "aet", "ant", "aet", "ant", "abt"]

- Add the words in their respective buckets using a hashtable `unordered_map<string, vector<string>>`  
hash["aet"] = "eat", "tea", "ate"  
hash["ant"] = "tan", "nat"  
hash["abt"] = "bat"
- Go over the bucket and add to the results

# Code – 49. Group Anagrams

Medium



LeetCode

[leetcode.com/problems/group-anagrams](https://leetcode.com/problems/group-anagrams)

**Code** Time:  $O(n * k \log k)$  Space:  $O(n * k)$  where  $n$  is the number of strings in `strs` and  $k$  is the maximum length of a string in `strs`

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    // go over the strs
    // sort each of them, store it
    // ["eat","tea","tan","ate","nat","bat"]
    // ["aet","aet","ant","aet","ant","abt"]
    // hash["aet"] = ["eat","tea","ate"]
    unordered_map<string, vector<string>> hash;
    for (const auto& s : strs) {
        string key = s;
        sort(key.begin(), key.end());
        hash[key].push_back(s);
    }
    // go over this hash map and push to the final output
    vector<vector<string>> result;
    for (const auto& [k, v] : hash) {
        result.push_back(v);
    }
    return result;
}
```

# Problem – Valid Parentheses

Easy



LeetCode

[leetcode.com/problems/valid-parentheses](https://leetcode.com/problems/valid-parentheses)

## Problem Statement

- You are given a string containing only the characters '(', ')', '{', '}', '[' and ']'
- A valid input have closed brackets by its own type

- **Example**

`()[]{} → valid`

`[]{}( → invalid`

`{()} → valid`

# Solution – Valid Parentheses

Easy



LeetCode

[leetcode.com/problems/valid-parentheses](https://leetcode.com/problems/valid-parentheses)

## Solution

- Loop through the string
- If **open** brackets (**{** push to a stack
- If **closed** brackets:
  - **pop** the last added bracket
  - **check** if the **closed** bracket corresponds to the **popped** bracket
  - if not, return false
- after the loop, **return true** if the **size** of the stack is empty (all brackets closed)

# Code – Valid Parentheses

Easy



LeetCode

[leetcode.com/problems/valid-parentheses](https://leetcode.com/problems/valid-parentheses)

## Code Time: $O(n)$ Space: $O(n)$

```
bool isValid(string s) {
    // stack (LIFO)
    std::stack<char> brackets;
    // O(n)
    for (int i = 0; i < s.size(); ++i) {
        char bracket = s[i];
        if (bracket == '(' || bracket == '[' || bracket == '{') {
            brackets.push(bracket);
        } else {
            if (brackets.size() == 0) return false;
            char lastBracket = brackets.top();
            if (bracket == ')' && lastBracket != '(') return false;
            if (bracket == '}' && lastBracket != '{') return false;
            if (bracket == ']' && lastBracket != '[') return false;
            brackets.pop();
        }
    }
    // all brackets must be closed
    return brackets.size() == 0;
}
```

# Problem – 125. Valid Palindrome

Easy



LeetCode

[leetcode.com/problems/valid-palindrome](https://leetcode.com/problems/valid-palindrome)

## Problem

- You are given a **string s**
- Return **true** if it is a palindrome
- Note that the string may contain **non-alphanumeric characters** that should be ignored and **uppercase/lowercase** that must be considered the same

- **Example:**

input = "A man, a plan, a canal: Panama"

output = true

after removing non-alphanumeric characters (including spaces) and turning everything into lowercase (or uppercase), the resulting string is a palindrome

# Solution – 125. Valid Palindrome

Easy



LeetCode

[leetcode.com/problems/valid-palindrome](https://leetcode.com/problems/valid-palindrome)

## Solution

- **Remove non-alphanumeric** characters:

```
auto end = remove_if(s.begin(), s.end(), [](char& c) {  
    return !isalnum();  
});
```

```
s.erase(end, s.end());
```

`remove_if` logically moves everything to the end of the string and return the iterator. Then, erase remove from the result of the iterator to the end of the strong

- **Transform** the string to lowercase:

```
transform(s.begin(), s.end(), s.begin(), [](char& c) {  
    return tolower(c);  
});
```

1<sup>st</sup> argument = beginning of string

2<sup>nd</sup> argument = end of the string

3<sup>rd</sup> argument = destination

4<sup>th</sup> argument = lambda function



# Solution – 125. Valid Palindrome

Easy



LeetCode

[leetcode.com/problems/valid-palindrome](https://leetcode.com/problems/valid-palindrome)

## Solution

- Have two pointers:

```
left = 0
```

```
right = s.size() - 1
```

- Loop incrementing left and decrementing right, checking the characters from both sides
- If they differ, **return false**
- At the end, **return true**

# Code – 125. Valid Palindrome

Easy



LeetCode

[leetcode.com/problems/valid-palindrome](https://leetcode.com/problems/valid-palindrome)

**Code** Time: **O(n)** Space: **O(1)**

```
bool isPalindrome(string s) {
    // transform everything into lowercase:
    // transform(begin, end, output begin)
    transform(s.begin(), s.end(), s.begin(), [](char& c) {
        return tolower(c);
    });

    // remove_if move everything that matches in the lambda
    // to the end of
    auto end = remove_if(s.begin(), s.end(), [](char& c) {
        return !isalnum(c);
    });
    s.erase(end, s.end());

    int left = 0;
    int right = s.size() - 1;

    while (left <= right) {
        if (s[left] != s[right]) return false;
        left++;
        right--;
    }
    return true;
}
```

# Problem – Minimum Number of Increments on Subarrays

Hard



LeetCode

[leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array](https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array)

## Problem Statement

- You are given an array of integers initialized with zeros (e.g. **[0,0,0,0]**)
- The goal is to reach some target (e.g. **[1, 2, 2, 3]**)
- The valid operations is to increment a subarray by one
- The output is the total number of operations

In this case:

**[1,1,1,1]** → increment the subarray starting from 0 to total size

**[1,2,2,2]** → increment the subarray starting from 1 to total size

**[1,2,2,3]** → increment the subarray starting and ending from the last element

**Output:** 3 (total number of operations)

# Solution – Minimum Number of Increments on Subarrays

Hard



LeetCode

[leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array](https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array)

## Solution

- Take this example:

`target = [1000, 1, 1000]`

- Initialize total number of operations `totalOp = target[0] = 1000`
- Loop through the array, compare the first element with the previous:  
`target[1] > target[0] → 1 > 1000` → do nothing, `totalOp` is still 1000
- `target[2] > target[1] → 1000 > 1` → add the difference to `totalOp`:  
 $\text{difference} = 1000 - 1 = 999$   
 $\text{totalOp} = 1000 + 999 = 1999$
- This is the number of operations needed, equivalent to:
  - add 1 to each element: `[1,1,1]`
  - add 999 to the subarray `[0,0]`
  - add 999 to the subarray `[2,2]`

# Code – Minimum Number of Increments on Subarrays

Hard



LeetCode

[leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array](https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array)

## Code

```
int minNumberOperations(vector<int>& target) {  
    int totalOp = target[0];  
    for (int i = 1; i < target.size(); ++i) {  
        // can't reuse  
        if (target[i - 1] < target[i]) {  
            totalOp += target[i] - target[i - 1];  
        }  
    }  
    return totalOp;  
}
```

# Code [2] – Minimum Number of Increments on Subarrays

Hard



LeetCode

[leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array](https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array)

## Code (optimized)

```
int minNumberOperations(vector<int>& target) {  
    return target[0] +  
        inner_product(target.begin() + 1, target.end(),  
            target.begin(), 0,  
            plus<int>(),  
            [](int curr, int prev) { return max(curr - prev, 0); });  
}
```

# Problem – 788. Rotated Digits

Medium



LeetCode

[leetcode.com/problems/rotated-digits](https://leetcode.com/problems/rotated-digits)

## Problem

- You are given a number **n**
- From the range between 1 to n, find “good” numbers
- A good number must meet **2 requirements**:
  - 1.** Be still valid after flipping: You physically “rotate” this number by 180 degrees, flip the number upside-down 2. The number can be either valid or invalid. For example, flipping **8** is still **8**, flipping **6** becomes **9**, but flipping **3**, becomes **ε** which is invalid.
  - 2.** Be a different digit after flipping. If you flip **1**, it is still a valid number but it is the same number (1), so it is not good. However, **16** is valid because it becomes a different number: **19**
- Return the the number of good numbers between **1** and **n**

# Problem – 788. Rotated Digits

Medium



LeetCode

[leetcode.com/problems/rotated-digits](https://leetcode.com/problems/rotated-digits)

## Solution

- The simplest and readable approach:
- Create a function to check if a number is good or not
- Go over the range (1,n) and check every number. If it is good, count as a valid
- Inside the function to check:
- Extract digit by digit from the number (digit = num % 10)
- Check if the digit is valid (a.k.a “flippable”). In other words, return false if it is 3, 4 or 7.
- Now check the second condition (same number). So keep a bool “changed”, if you find a number that “changes”, mark changed as true. The numbers are 2, 5, 6 and 9, since when they flip they become different numbers
- Return “changed”



# Problem – 788. Rotated Digits

Medium



LeetCode

[leetcode.com/problems/rotated-digits](https://leetcode.com/problems/rotated-digits)

**Code** Time:  **$O(n \log n)$**  Space:  **$O(1)$**

For each number, we examine each of its digits:

- A number  $i$  has  $\log_{10}(i)$  digits  $\rightarrow$  in worst case:  $O(\log n)$  per number

```
int rotatedDigits(int n) {
    int count = 0;
    for (int i = 1; i <= n; ++i) {
        if (isGood(i)) count++;
    }
    return count;
}

bool isGood(int num) {
    bool changed = false;
    while (num > 0) {
        int digit = num % 10;
        if (digit == 3 || digit == 4 || digit == 7) return false;
        if (digit == 2 || digit == 5 || digit == 6 || digit == 9)
            changed = true;
        num /= 10;
    }
    return changed;
}
```

# Problem – 383. Ransom Note

Easy



LeetCode

[leetcode.com/problems/ransom-note](https://leetcode.com/problems/ransom-note)

## Problem

- You are given two strings: **magazine** and **ransomNote**
- Return true if **ransomNote** can be constructed by using letters from **magazine**
- A letter **cannot** be reused

- **Example:**

ransomNote = "aa", magazine = "ab"

**Output:** false (a letter from magazine cannot be used twice)

ransomNote = "aa", magazine = "aab"

**Output:** true

# Solution – 383. Ransom Note

Easy



LeetCode

[leetcode.com/problems/ransom-note](https://leetcode.com/problems/ransom-note)

## Solution

- Initialize an array with 26 characters (total letters in the English alphabet)
- Go over **magazine** string and count each character
- Go over **ransomNote** string and decrease each character
- If you get a negative number, return false

# Code – 383. Ransom Note

Easy



LeetCode

[leetcode.com/problems/ransom-note](https://leetcode.com/problems/ransom-note)

## Code

Time:  $O(n + m)$  Space:  $O(k)$  where  $n$  is the length of **magazine** and  $m$  the length of **ransomNote**, and  $k$  is the number of unique characters in magazine

```
bool canConstruct(string ransomNote, string magazine) {  
    int count[26] = {0};  
    for (const char& c : magazine) {  
        count[c - 'a']++;  
    }  
    for (const char& c : ransomNote) {  
        if (--count[c - 'a'] < 0) return false;  
    }  
    return true;  
}
```

# Problem – 8. String to Integer (atoi)

Medium



LeetCode

[leetcode.com/problems/string-to-integer-atoi](https://leetcode.com/problems/string-to-integer-atoi)

## Problem

- You are given a string **s**
- Implement **myAtoi(string s)** using the following rules:
- Skip leading whitespace
- Determine sign (+ or -)
- Convert digits until non-digit or end of string
- Clamp result to 32-bit signed integer range  $[-2^{31}, 2^{31} - 1]$

# Solution – 8. String to Integer (atoi)

Medium



LeetCode

[leetcode.com/problems/string-to-integer-atoi](https://leetcode.com/problems/string-to-integer-atoi)

## Solution

- Initialize an index  $i$
- Position  $i$  to skip white spaces
- Check the sign and set a variable  $sign = -1$  or  $1$
- Go over the remaining of the string and use the following:  
 $digit = digit * 10 + s[i] - '0'$
- **Important:** use long long for the result and check overflows:  
 $if (sign == 1 \ \&\& \ result > INT\_MAX) \ return \ INT\_MAX;$   
 $if (sign == -1 \ \&\& \ -result < INT\_MIN) \ return \ INT\_MIN;$

# Problem – 8. String to Integer (atoi)

Medium



LeetCode

[leetcode.com/problems/string-to-integer-atoi](https://leetcode.com/problems/string-to-integer-atoi)

**Code** Time: **O(n)** Space: **O(1)**

```
int myAtoi(string s) {
    int i = 0;
    int n = s.size();
    // skip leading whitespace
    while (i < n && s[i] == ' ') i++;

    // some sanity check
    if (i == n) return 0;

    // check the sign
    int sign = 1;
    if (s[i] == '-') {
        sign = -1;
        i++;
    } else if (s[i] == '+') {
        // keep sign = 1
        i++;
    }

    // convert
    long long result = 0;
    while (i < n && isdigit(s[i])) {
        result = result * 10 + s[i] - '0';
        if (sign == 1 && result > INT_MAX) return INT_MAX;
        if (sign == -1 && -result < INT_MIN) return INT_MIN;
        ++i;
    }
    return result * sign;
}
```