LeetCode 155 — Min Stack

🔗 https://leetcode.com/problems/min-stack/

1. Problem Title & Link

Title: LeetCode 155: Min Stack
Link: https://leetcode.com/problems/min-stack/

2. Problem Statement (Short Summary)

Design a stack that supports:

1. push(x)
2. pop()
3. top()
4. getMin() → returns the **minimum element** in constant time.

All operations **must run in O(1) time**.

3. Examples (Input → Output)

Input:

push(-2)

push(0)

push(-3)

getMin() → -3

pop()

top() → 0

getMin() → -2

4. Constraints

- Up to **3 * 10^4** operations
- Must achieve **O(1) time per operation**
- Easy to think but tricky edge case → keep min on pop as well

5. Core Concept (Pattern / Topic)

⭐ **Stack + Auxiliary Min Tracking**

Techniques:

✔ Two-stack method

✔ Single-stack with paired values

✔ Optional optimized monotonic approach

6. Thought Process (Step-by-Step)

❌ WRONG brute idea

Keep full list → getMin = O(n) scan → violates O(1).

✔ Correct idea

Maintain current **minimum at each stack state**.

Approach A — Two stacks

- Main stack → store values
- Min stack → store current minimum at each push
- If value ≤ currentMin → push to minStack
- On pop, if popped == minStack top → pop minStack too

Approach B — Single stack storing pairs (value, currentMin)

- Push tuple (x, min(x, currentMin))
- getMin just returns stored minimum on top
- Cleanest implementation

7. Visual Diagram

Push sequence → stack + min:

Operation	Stack	MinStack
push -2	-2	-2
push 0	-2,0	-2
push -3	-2,0,-3	-2,-3

On pop -3 → min reverts automatically to -2.

8. Pseudocode

```
stack = []
minStack = []

push(x):
    stack.push(x)
    if minStack empty or x <= minStack.top():
        minStack.push(x)

pop():
    removed = stack.pop()
    if removed == minStack.top():
        minStack.pop()
```

```
top():
    return stack.top()

getMin():
    return minStack.top()
```

9. Code Implementation

✅ Python (Two-stack method)

```python
class MinStack:

    def __init__(self):
        self.stack = []
        self.minStack = []

    def push(self, val: int) -> None:
        self.stack.append(val)
        if not self.minStack or val <= self.minStack[-1]:
            self.minStack.append(val)

    def pop(self) -> None:
        val = self.stack.pop()
        if val == self.minStack[-1]:
            self.minStack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.minStack[-1]
```

✅ Java (Two-stack method)

```java
class MinStack {

    Stack<Integer> stack;
    Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }
```

```
    public void push(int val) {
        stack.push(val);
        if (minStack.isEmpty() || val <= minStack.peek())
            minStack.push(val);
    }

    public void pop() {
        int removed = stack.pop();
        if (removed == minStack.peek())
            minStack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

10. Time & Space Complexity

Operation	Time	Space
push	O(1)	O(n)
pop	O(1)	O(n)
top	O(1)	O(1)
getMin	O(1)	O(1)

Space used to keep historical minimums.

11. Common Mistakes / Edge Cases

❌ Forgetting to pop from minStack when popping same value

❌ Only storing global min instead of min per level

❌ Not handling empty stack properly

Edge cases:

✔ push duplicate mins
✔ pop until empty
✔ getMin immediately after a pop

12. Detailed Dry Run

Commands:

push(-2)

push(0)

push(-3)

getMin() → -3

pop()

top() → 0

getMin() → -2

Step	Stack	MinStack
push -2	[-2]	[-2]
push 0	[-2,0]	[-2]
push -3	[-2,0,-3]	[-2,-3]
getMin	-3	top→-3
pop	[-2,0]	[-2]
top	0	
getMin	-2	top→-2

13. Common Use Cases

- O(1) min lookup in stack-based algorithms
- Stock/temperature minimum queries
- Sliding computations inside stack operations
- Classic interview LLD question — demonstrates auxiliary stack concept

14. Common Traps

⚠ Forgetting equal case on push (val <= min)

⚠ Trying to pop without checking empty

⚠ Storing only values without tracking min at each stage

15. Builds To (Related LeetCode)

- LC 716 **Max Stack**
- LC 295 **Median from Data Stream**
- LC 239 **Sliding Window Maximum**
- LC 503 **Next Greater Element II**

16. Alternate Approaches + Comparison

Method	Good?	remark
Two stacks	⭐ Recommended	Clean + always O(1)
Single stack of pairs `(val,min)`	Also clean	No second stack needed
Track min diff encoding	Space optimized	Harder debug, less readable

17. Why Solution Works

Because at **every push**, we store the **minimum so far**, allowing getMin() to return the min for current stack state instantly in O(1) without searching.

18. Variations / Follow-Up

- Support getMax() → similar using second maxStack
- Support min in constant **space** per value using difference encoding
- Design **MinQueue**, **MinDeque** (via two MinStacks)
- Add getMedian() or popMin() operations — advanced design

SINGLE STACK (Value + Min Pair) Approach

Core Idea

Instead of using **two stacks**, we store **both the value and the minimum at this point in time** together in *one stack*.

So each push maintains:

stack.push((value, minSoFarAtThisPoint))

That means:

- top().min → gives min instantly
- No separate min stack required
- Popping automatically restores previous min

📌 How It Works

Push Value	Stack Stores As (value, minSoFar)
push 5	(5,5)
push 2	(2,2) ← new min
push 4	(4,2) ← min stays 2
push 1	(1,1) ← new min

- getting min = stack.top().min
- pop removes entire pair, so min auto-updates

Python — Single Stack Implementation

```
class MinStack:

    def __init__(self):
        self.stack = []  # each entry → (value, min_so_far)

    def push(self, val: int) -> None:
        if not self.stack:
            self.stack.append((val, val))
        else:
            currentMin = self.stack[-1][1]
            self.stack.append((val, min(val, currentMin)))

    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0]
```

```
    def getMin(self) -> int:
        return self.stack[-1][1]
```

Java — Single Stack Implementation

```
class MinStack {
    private Stack<int[]> stack;  // each element = [value, minSoFar]

    public MinStack() {
        stack = new Stack<>();
    }

    public void push(int val) {
        if (stack.isEmpty())
            stack.push(new int[]{val, val});
        else {
            int currentMin = stack.peek()[1];
            stack.push(new int[]{val, Math.min(val, currentMin)});
        }
    }

    public void pop() {
        stack.pop();
    }

    public int top() {
        return stack.peek()[0];
    }

    public int getMin() {
        return stack.peek()[1];
    }
}
```

Complexity

Operation	Time	Space
push	O(1)	O(n)
pop	O(1)	O(1)
getMin	O(1)	O(1)

Pros vs Two-Stack Approach

Feature	Two-Stack	Single-Stack (Pairs)
Space usage	Slightly higher	More compact
Min tracking	Clear/explicit	Embedded in each element
Code simplicity	Moderate	Very clean & elegant
Teaching clarity	High	High — best for students

One-Line Intuition

Every element remembers the minimum at the time it was pushed —
so popping rewinds history automatically.