Week 2 Worked Companion

Choosing state from the contract

Problem	Stored state	Update justification	Cost
Longest unique interval	Set, left boundary	Remove repeat before insertion	Expected O(n) time
Shortest target interval	Sum, left boundary	Positivity and recorded lengths	O(n) time
Min Stack	Value and minimum per depth	Restore previous minimum	O(1) queries
Postfix evaluation	Stack of subexpression values	Pop right, then combine	O(t) time

Distinguish pointer movements, set operations, and list resizing when counting work. Explain both a successful example and a counterexample that breaks an incorrect state update.

3 · Staged hints

Return the longest substring length without repeats; spaces count, empty input gives 0.

LeetCode 3: Longest Substring Without Repeating Characters · Medium

- H1. Are all characters in the current interval distinct?
- H2. How many left moves are needed to remove a duplicate?
- H3. Remove leftmost characters until the incoming character is absent, then add it.

`text="abba c"`

right, character	left before	Removed	Window after insertion	best
0, a	0	None	a	1
1, b	0	None	ab	2
2, b	0	a, b	b	2
3, a	2	None	ba	2
4, space	2	None	ba SP	3
5, c	2	None	ba SP c	4

3 · Correctness and cost

After insertion, the set represents the unique-character window. Its left boundary is the earliest feasible one for that right endpoint. Each pointer advances at most n times.

right advances n times and left at most n times without moving back. Sum their movements, at most 2n, instead of multiplying apparent loop bounds. With expected O(1) set operations, time is expected O(n). At most min(n,a) characters are retained, giving O(min(n,a)) auxiliary space for alphabet size a.

3 · Counterexamples and checks

A single removal on “abba” leaves b duplicated. Empty text gives 0; one space gives 1.

```
text="abba c" → 4 ("ba c")
```

3 · Complete implementation

```
def longest_unique(text):
    active, left, best = set(), 0, 0
    for right, char in enumerate(text):
        while char in active:
            active.remove(text[left])
            left += 1
        active.add(char)
        best = max(best, right - left + 1)
    return best
```

Remove the character at the old left boundary and advance left until the duplicate is gone. At the second b in `abba`, removing only a leaves b duplicated, so one if is insufficient. Measure length after inserting the incoming character.

209 · Staged hints

Return the minimum length of a contiguous subarray of positive numbers with sum at least positive target. Return zero if none exists.

LeetCode 209: Minimum Size Subarray Sum · Medium

- H1. Must every interval be summed again?
- H2. How does removing the left value change the sum?
- H3. Extend right, then record and shrink while the sum qualifies.

`target=6, nums=[1,2,4,2]`

Action	Window	total	best
Add 1 and 2	[1,2]	3	Not found
Add 4, record length	[1,2,4]	7	3
Remove 1, record length	[2,4]	6	2
Remove 2, stop shrinking	[4]	4	2
Add final 2, record	[4,2]	6	2

209 · Correctness and cost

Positivity makes shrinking decrease the sum. Record each valid window before shrinking. A discarded left boundary can only produce longer later windows.

Each value is added once and removed at most once. Total boundary movement is at most 2n, giving O(n) time. No window is copied; total, left, right, and best use O(1) auxiliary space. The n-entry input is counted separately from workspace.

209 · Counterexamples and checks

Outside the positive-only contract, target=3, [1,-1,3] can make this rule miss length 1.

```
target=6, nums=[1,2,4,2] → 2
```

209 · Complete implementation

```
def min_subarray(target, nums):
    total = left = 0
    best = len(nums) + 1
    for right, value in enumerate(nums):
        total += value
        while total >= target:
            best = min(best, right - left + 1)
            total -= nums[left]
            left += 1
    return 0 if best > len(nums) else best
```

Add the incoming value first. While the target is met, record the current length and remove the left value to try a shorter interval. best=n+1 marks no solution yet, since a valid interval can be at most n long.

155 · Staged hints

push adds a value and pop removes the top. top returns the top value; getMin returns the minimum of all retained values. Queries and removals are called only on nonempty stacks.

LeetCode 155: Min Stack · Medium

- H1. How can the previous minimum be restored after a pop?
- H2. What happens when the minimum occurs twice?
- H3. Store (value, minimum through this depth) per entry. Create entries separately for each instance.

`push(2), push(2), push(5), pop(), pop()`

Operation	Stored (value, minimum) pairs	getMin result
push(2)	[(2,2)]	2
push(2)	[(2,2),(2,2)]	2
push(5)	[(2,2),(2,2),(5,2)]	2
pop()	[(2,2),(2,2)]	2
pop()	[(2,2)]	2

155 · Correctness and cost

Induction on depth gives the correct prefix minimum; popping the last pair restores the previous prefix minimum.

At depth d, getMin and top read one slot and cost O(1). Push and pop are amortized O(1), including list resizing. Two values per depth give 2d stored values, or O(d) space. Recomputing min over the whole stack would make each query O(d).

155 · Counterexamples and checks

Calling min over the stack costs O(d). A single current minimum loses its predecessor after a pop.

`push(2),push(2),push(5),pop(),pop(),getMin() → 2`

155 · Complete implementation

```
class MinStack:
    def __init__(self):
        self.entries = []

    def push(self, val):
        minimum = min(val, self.entries[-1][1]) if self.entries else val
        self.entries.append((val, minimum))

    def pop(self):
        self.entries.pop()

    def top(self):
        return self.entries[-1][0]

    def getMin(self):
        return self.entries[-1][1]
```

Each tuple stores `(current value, minimum through this depth)`. Removing the last tuple reveals the previous depth's minimum. getMin reads one slot in the last tuple instead of scanning all values with min.

In `entries[-1][1]`, −1 selects the last tuple and 1 selects its second field. For `[(4,4),(7,4)]`, top is 7 and getMin is 4. On an empty list the conditional expression evaluates only its else branch, val, so it never reads a nonexistent last entry.

150 · Staged hints

Evaluate valid postfix string tokens. Integer division truncates toward zero; inputs never divide by zero.

LeetCode 150: Evaluate Reverse Polish Notation · Medium

- H1. Is the first popped value the left or right operand?
- H2. Which way should -5/2 be truncated?
- H3. Push numbers; pop right then left for an operator and push its result. Divide absolute integers and restore the sign.

`tokens=["9","4","-","-2","/"]`

Token	Stack before	Operation	Stack after
9	[]	push(9)	[9]
4	[9]	push(4)	[9,4]
-	[9,4]	left=9, right=4	[5]
-2	[5]	push(-2)	[5,-2]
/	[5,-2]	5 / -2, truncate toward zero	[-2]

150 · Correctness and cost

The stack preserves completed subexpression values in order. Each operator combines the final two; a valid complete expression leaves one value.

Each of t tokens performs at most two pops, one append, and one arithmetic operation, giving O(t) time. The bounded 32-bit intermediates support constant-cost arithmetic. Many uncombined operands can accumulate, so peak auxiliary space is O(t), even though only one value remains at the end.

150 · Counterexamples and checks

5//-2 is -3, but truncation toward zero gives -2. Reversing operands changes subtraction.

`tokens=["9","4","-","-2","/"] → -2`

`(left < 0) != (right < 0)` tests whether the signs differ. For 5 and −2, False != True gives a negative result, −2. For −5 and −2, True != True is False, giving +2. Operator token `"-"` and numeric token `"-2"` are different strings, so the latter is converted with int.

150 · Complete implementation

```
def rpn(tokens):
    stack = []
    for token in tokens:
        if token not in {"+", "-", "*", "/"}:
            stack.append(int(token))
            continue
        right, left = stack.pop(), stack.pop()
        if token == "+":
            value = left + right
        elif token == "-":
            value = left - right
        elif token == "*":
            value = left * right
        else:
            quotient = abs(left) // abs(right)
            value = -quotient if (left < 0) != (right < 0) else quotient
        stack.append(value)
    return stack[0]
```

At an operator, the first popped value is the right operand and the second is the left. Push their result back. For division, divide magnitudes and restore the sign to truncate toward zero.

Session 3 answer checks

1. The last fixed-width sum is 6. At right=3, remove index 1 and add index 3.
2. On the second b in “abba”, removing only a leaves the earlier b. A set cannot represent the repeated b in that interval.
3. Both pointers move only forward, at most n positions each. No iteration resets the left pointer.
4. The positive-only rule returns 3; the true shortest length is 1. Removing -1 would increase the sum, so the stopping argument fails.

Session 4 answer checks

1. The minimum is 2; the first (2,2) pair remains after the second is removed.
2. Push 2 into one instance and 7 into another. Their top values must be 2 and 7, and the second minimum must be 7. Shared state fails these checks.
3. The first pop returns the right operand, 4. The subtraction is 9-4.
4. Python floor division gives -3; truncation toward zero gives -2.
5. Scanning all d entries costs O(d), replacing a direct O(1) lookup.

Execution and sources

```
cd code
python demo_03.py
python demo_04.py
python verify.py
```

Completed implementations are in `solutions.py`; independent work is in `student_week_02.py`. On LeetCode, adapt each function to the official method signature.

Erik Demaine and Srini Devadas, Introduction to Algorithms, MIT 6.006 (Fall 2011).