Week 2: Sliding windows and stacks

Contiguous intervals in event logs

In the event log `"abba c"`, find the longest contiguous interval containing no repeated character. A space counts as a character, so `"ba c"` has length 4.

Rebuilding the character collection for every interval repeatedly reads the same records. Can we update only the entering and leaving characters as the boundaries move?

This week: incremental window state and stacks that retrieve the most recently stored information first.

Week 2 problems and learning outcomes

Session 3: fixed-width and variable-width intervals. Session 4: per-depth minima and postfix evaluation order.

- Window boundaries and stored state
- Total pointer movements and time cost
- Restored stack state and operand order
- Positivity, repeated minima, and negative division checks

Practice uses LeetCode 3, 209, 155, and 150.

Updating a fixed-width window

For `nums=[1,2,4,2]` and width `w=2`, the contiguous window sums are `[3,6,6]`. Initialize with `sum(nums[:w])`.

New right index	Leaving value	Entering value	Updated sum
Initial interval 0..1	None	1, 2	3
2	nums[0] = 1	nums[2] = 4	3 - 1 + 4 = 6
3	nums[1] = 2	nums[3] = 2	6 - 2 + 2 = 6

New right index `r`: leaving index `r-w`. The range `range(w, len(nums))` includes the final index.

Fixed-width implementation and cost

```
def fixed_window_sums(nums, width):
    if not 1 <= width <= len(nums):
        raise ValueError("width must be within 1..len(nums)")
    total = sum(nums[:width])
    result = [total]
    for right in range(width, len(nums)):
        total += nums[right] - nums[right - width]
        result.append(total)
    return result
```

With constant-cost integer arithmetic, time is O(n). The initial slice uses O(w) temporary space; the output uses O(n-w+1) space. `total += value` accumulates; `total =+ value` assigns the value with unary plus.

Subtract the one outgoing value and add the one incoming value. `right-width` is the outgoing position. Slicing and summing a width-sized interval every time would take O(n·width); this code sums only the initial interval.

Fixed and variable window boundaries

A fixed-width window moves both boundaries by one position. A variable-width window moves its left boundary according to the problem condition.

Problem	Stored information	When to advance left
Fixed-width sum	Current sum	Once per incoming value
Longest unique interval	Current character set	While the incoming character is present
Shortest interval reaching a target	Current sum	While the sum reaches the target

Inclusive endpoints: `left`, `right`. Length: `right-left+1`. Shrinking can leave an empty interval.

LeetCode 3: contract and baseline

Longest Substring Without Repeating Characters asks for the maximum length of a contiguous substring with no repeated character. A subsequence may skip positions; a substring cannot.

Input: 0 to 100000 characters, including letters, digits, symbols, and spaces. Empty text: 0.

A baseline starts an empty set at every left endpoint and scans right until a repeat appears. This revisits intervals and performs $O\left(n^2\right)$ character checks in the worst case.

Invariant of the unique-character window

Before inserting the next character, `active` contains the distinct characters in `text[left:right]`. If the incoming `char` is already present, remove `text[left]` and increment `left`.

Repeat until the duplicate is gone, then insert `char`. The set now represents the inclusive interval `text[left:right+1]`, which has no repeated character.

Update the best length after insertion. When advancing left, remove the character at the old left boundary, not one outside the window or the incoming character.

Tracing LeetCode 3

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

The symbol `SP` marks a space. Inserting the third character requires two removals, so a single `if` is insufficient.

Implementing the longest unique interval

```
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
```

`"abba"` returns 2, empty text returns 0, and one space returns 1. To return a substring rather than its length, also retain the starting position of the best interval.

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.

Correctness and total pointer movements

A window containing the previous occurrence cannot include the incoming duplicate. Move left only until that occurrence is excluded, giving the longest feasible window for each right endpoint. Earlier discarded starts still contain a repetition and cannot become feasible again.

The right pointer advances n times and the left pointer at most n times. The nested `while` therefore executes at most n times in total.

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.

LeetCode 209: target sum and positivity

Minimum Size Subarray Sum asks for the shortest contiguous interval whose sum is at least `target`. Return 0 when none exists.

Input length: 1 to 100000. Each value: 1 to 10000. `target`: 1 to 10^9. All values and the target are positive.

For `target=6` and `nums=[1,2,4,2]`, either `[2,4]` or `[4,2]` gives length 2. Accumulating sums separately from every starting position takes $O\left(n^2\right)$ time in the worst case.

Recording and shrinking valid intervals

`total` is the current window sum. Add the rightmost value, then record the length and remove the leftmost value while the target is met.

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

Finally remove 4; the sum becomes 2, so shrinking stops. Record the length before removing a value.

Implementing the shortest interval

```
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
```

`best=n+1`: no valid interval found yet

With `target=8` and `[1,2,4]`, the total is insufficient, so the result is 0.

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.

Shortest-interval correctness and cost

Adding a positive value increases the sum; removing one decreases it. If the sum is below the target, further shrinking cannot produce a valid interval.

When the sum qualifies, record the length before shrinking. Every discarded start already produced a recorded valid interval. Reusing that start at a later endpoint only makes a longer interval, so no better answer is lost.

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.

A counterexample with negative values

Remove the official positivity condition and try `target=3`, `nums=[1,-1,3]`.

The total reaches 3 and records length 3. Removing the leading 1 reduces the sum to 2, stopping the loop. It misses the actual answer `[3]` of length 1.

Removing the next value, -1, would raise the sum back to 3. Removing a negative value can increase the sum, invalidating the monotonicity argument. This example is outside the official input contract and explains why positivity matters.

Session 3 concept checks

1. For `nums=[1,2,4,2]` and width 2, what is the last window sum? Give the leaving index as well.
2. If `while` becomes `if` on `"abba"`, where does the state first become invalid?
3. Why does the inner `while` execute only O(n) times overall?
4. For `target=3`, `[1,-1,3]`, compare the positive-only algorithm's result with the actual answer.

Undo operations and recoverable stack state

After two edits, one undo reverses the most recent edit first. This is last-in, first-out (LIFO) order.

A record tool displaying the current minimum must also restore the previous minimum after removing the newest value. Retaining only one current minimum is insufficient when that value is removed.

Session 4 designs auxiliary state stored with each value and uses the same retrieval order to evaluate postfix expressions.

Stack operations with Python lists

```
stack = []
stack.append(9)
stack.append(4)
print(stack[-1])     # 4: inspect without removal
right = stack.pop()  # 4
left = stack.pop()   # 9
```

Use the list end as the top. `append` and end `pop` are amortized O(1); `stack[-1]` is O(1). Removing index 0 shifts later entries and costs O(d).

Calling `pop()` or `[-1]` on an empty list raises an exception. Here d is current depth; amortized cost accounts for total work over a sequence of operations.

LeetCode 155: minima at each depth

Min Stack requires constant-time `push`, `pop`, `top`, and `getMin`. Values are signed 32-bit integers, with at most 30000 calls. `pop`, `top`, and `getMin` are called only on nonempty stacks.

Computing `min(stack)` at every query costs O(d). Instead, store `(value, minimum through this depth)` at each depth.

The new minimum is `min(new value, previous minimum)`. Removing the last pair reveals the previous minimum in the pair below it.

Tracing repeated minima

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

Each entry retains the minimum at its own depth. Removing one occurrence of 2 leaves the earlier occurrence, so the minimum remains 2.

Min Stack object and 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.

Independent instance state and self

Create a new `self.entries` list when each object is constructed. In `first.push(2)`, Python passes `first` as the method's `self` argument.

```
first = MinStack()
second = MinStack()
first.push(2)
second.push(7)
print(first.getMin(), second.getMin())  # 2 7
```

A shared `entries=[]` in the class body can mix values from separate stacks. An ordinary function receives no instance automatically.

Min Stack correctness and cost

The first entry’s minimum is its own value. If the previous prefix minimum is correct, taking the smaller of it and the new value gives the correct new minimum. A pop removes only the final entry and reveals the unchanged previous prefix.

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).

LeetCode 150: postfix notation and subexpressions

Evaluate Reverse Polish Notation evaluates expressions whose operators follow their operands. `9 4 - -2 /` means `(9-4)/(-2)` and returns -2.

There are 1 to 10000 tokens. Operands are integers from -200 to 200; operators are `+ - * /`. The expression is valid, never divides by zero, and all intermediate results and the answer fit signed 32-bit integers.

- Number: push onto the stack
- Operator: pop right, then left; push the result

Tracing postfix evaluation

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]

The stack preserves completed subexpression values in their original order. For subtraction and division, distinguish pop order from operand order.

Truncation in negative division

The problem requires truncation toward zero. Python `//` rounds toward negative infinity, so negative quotients can differ.

Expression	Python //	Truncate toward zero
5 / 2	2	2
5 / -2	-3	-2
-5 / 2	-3	-2
-5 / -2	2	2

Compute the magnitude as `abs(left) // abs(right)` and negate it when the signs differ. This uses integer arithmetic without a floating-point conversion.

`(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.

Implementing postfix evaluation

```
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.

Subexpression invariant and evaluation cost

A number creates one completed subexpression. An operator combines the last two completed values in operand order. Repeating this process leaves one value for the entire valid expression.

`["9","4","-"]` returns 5; reversing the subtraction gives -5. Inputs are string tokens: `["-2"]` represents one number and returns -2.

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.

Session 4 concept checks

1. Explain the final result and retained state of `push(2), push(2), pop(), getMin()`.
2. Which inputs expose two `MinStack` objects accidentally sharing a list?
3. In `["9","4","-"]`, which operand is returned by the first `pop`?
4. Comparison: `5//-2` with the problem's required result for `5/-2`
5. How does calling `min(entries)` inside `getMin` change query cost?

Checks: the remaining tuple is (2,2), with minimum 2. Push into only one stack and check that the other stays empty to detect shared storage. The first pop returns 4. 5//−2 gives −3; the required result is −2. Recomputing min makes a query O(n).

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.

Independent exercises and submission

Start with the student exercises. Define inputs, outputs, and constraints before implementing `student_week_02.py`. Include the state meaning, correctness argument, time and space costs, and your own boundary inputs.

Week 2 code · Worked solutions

All Week 2 materials · Instructor guide

Core sources

Erik Demaine and Srini Devadas, Introduction to Algorithms, MIT 6.006 (Fall 2011). Study algorithmic state, correctness, and cost analysis together.

Use the official LeetCode pages for complete problem statements and constraints.

- 3: Longest Substring Without Repeating Characters
- 209: Minimum Size Subarray Sum
- 155: Min Stack
- 150: Evaluate Reverse Polish Notation