Week 3: Binary search and linked lists

Finding a measurement in ordered records

A device stores sorted measurements in rows. When a new reading arrives, can we find it without inspecting every record? Then a queue of records must be reordered while preserving the same node objects.

This week uses candidate intervals for search and changes links to reverse or merge lists.

Learning outcomes

Session 5: matrix and rotated-array search. Session 6: list reversal and sorted-list merge.

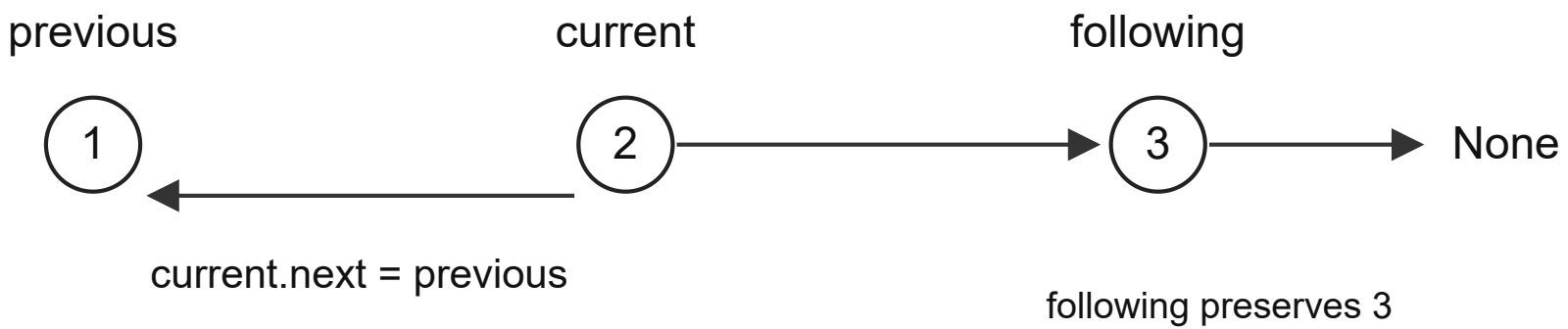
- Define candidate intervals and node references.
- Explain discarded candidates and preserved nodes.
- Trace updates, prove termination, and analyze cost.

Practice: LeetCode 74, 33, 206, 21.

Positions and node references

An array directly accesses position i by index. A linked list follows next references from its head. Two nodes with the same value 2 can still be distinct objects.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val, self.next = val, next
```



The diagram processes node 2. Preserve node 3 as following before changing 2's next to node 1. A node's value and its links are different data. `node = node.next` moves a local reference; `node.next = other` changes the structure.

74 · Idea and execution trace

LeetCode 74: Search a 2D Matrix · Medium

Return whether target occurs in the matrix.

Each row is sorted and its first value exceeds the previous row's last value. Reading rows consecutively therefore gives one sorted sequence. Search that virtual sequence without copying it; translate each position into a row and column.

`matrix=[[1,3,5],[7,9,11]], target=9`. Bounds are shown before each comparison.

left,right	mid	row,col	Value	Next action
0,5	2	0,2	5	5 < 9 → left=3
3,5	4	1,1	9	Return True

74 · Turning the idea into code

```
def search_matrix(matrix, target):
    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1
    while left <= right:
        mid = (left + right) // 2
        value = matrix[mid // cols][mid % cols]
        if value == target:
            return True
        if value < target:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

`mid // cols` counts complete rows before the position; `mid % cols` locates it within a row. Position 4 with three columns is row 1, column 1.

If the value is too small, discard through mid; if too large, discard from mid onward. Excluding the tested position guarantees progress even with one candidate left.

74 · Why it works; time and memory

The cross-row property makes the flattened order sorted. Comparisons eliminate an impossible half; every update shrinks the interval, and an empty interval means absence.

For $N=R\cdot C$ elements, comparisons reduce the candidates roughly to $N/2$, $N/4$, and so on. $N=8$ needs at most four comparisons; time is $O(\log(N+1))$. Index arithmetic and each list lookup cost $O(1)$.

No flattened list is built. Only a fixed number of indices and values are retained, so auxiliary space is $O(1)$. Copying the matrix into a new sequence would change this space bound.

Termination when the target is absent

Search for target=8 in `[[1,3,5],[7,9,11]]`. The interval `[left,right]` includes both remaining endpoints.

Interval before comparison	mid and value	Reason for removal	Next interval
[0,5]	2, value 5	All positions at or below 2 have values below 8	[3,5]
[3,5]	4, value 9	All positions at or above 4 have values above 8	[3,3]
[3,3]	3, value 7	The last candidate also fails	[4,3]

left>right means no candidate remains, so return False. left==right still requires one comparison. Updating left=mid can repeat mid=3 forever on `[3,4]`.

Halving N candidates takes about $\log_2 N$ reductions to reach one. Including its comparison, the worst count is $\text{floor}(\log_2 N)+1$ for $N \geq 1$.

33 · Idea and execution trace

LeetCode 33: Search in Rotated Sorted Array · Medium

Return target's index in a rotated sorted array, or -1.

Although a rotated array is not globally sorted, with distinct values at least one half around mid is sorted. Identify that half, then test whether target lies within its value range to decide which half to discard.

`nums=[4,5,6,7,0,1,2], target=0`

left,mid,right	Sorted half	Target test	Update
0,3,6	Left: 4..7	0 is outside	left=4
4,5,6	Left: 0..1	0 is inside	right=4
4,4,4	nums[4]=0	Match	Return 4

33 · Turning the idea into code

```
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

`nums[left] <= nums[mid]` identifies a sorted left half. Keep it only when `nums[left] <= target < nums[mid]`. Equality at mid was already tested, hence the strict inequality at that end.

The sorted-right case is symmetric. Sorting again would change the original positions that the function must return.

33 · Why it works; time and memory

Distinct values and one rotation ensure a sorted half. Range tests discard only an impossible half; mid±1 guarantees progress.

Each iteration retains only one half, excluding mid. There are O(log(n+1)) iterations, each with a fixed number of comparisons and lookups.

No new array or recursive call stack is created, so auxiliary space is O(1). With duplicates, an input such as `[1,1,1,0,1]` can have equal endpoints and midpoint, hiding which half is sorted. This code and bound depend on distinct values.

206 · Idea and execution trace

LeetCode 206: Reverse Linked List · Easy

Reverse a singly linked list and return its new head.

A linked list's order is determined by next links. Reverse each link without allocating replacement nodes. Save the original successor before changing the link, or the unprocessed remainder becomes unreachable.

Input: `1 → 2 → 3 → None`. Each row is the state after one loop body.

Node processed	Chain at previous	Chain at current
None yet	None	1 → 2 → 3
1	1 → None	2 → 3
2	2 → 1 → None	3
3	3 → 2 → 1 → None	None

206 · Turning the idea into code

```
def reverse_list(head):
    previous, current = None, head
    while current:
        following = current.next
        current.next = previous
        previous, current = current, following
    return previous
```

`following = current.next` preserves the successor. `current.next = previous` reverses one link. The final simultaneous assignment advances previous to the processed node and current to the saved successor.

When current is None, no node remains to process and previous is the new head. An empty input skips the loop and returns None.

While processing node 2, previous points to 1, current to 2, and following to 3. After reversing next, the simultaneous assignment first evaluates its right side `(current,following)` as `(2,3)`. previous therefore becomes 2 and current becomes 3. Advancing through `current.next` would follow the reversed edge back to 1, which is why the saved following is used.

206 · Why it works; time and memory

previous holds the reversed prefix and current the untouched suffix. Moving one node preserves all nodes; when current is None, previous is the full reversed list.

Each of n nodes is visited once, with a fixed number of reference assignments per visit, giving O(n) time. Reversing the first link does not reverse later links; every node must be processed.

Only previous, current, and following are added. Existing nodes are reused, so auxiliary space is O(1) and no output nodes are allocated. The O(n) input structure is distinct from extra workspace. A recursive implementation would need O(n) call-stack space.

21 · Idea and execution trace

LeetCode 21: Merge Two Sorted Lists · Easy

Splice two sorted linked lists into one sorted list.

The smallest remaining value must be at one of the two sorted-list heads. Attach the smaller head to the result tail and advance only that input. There is no need to search farther down either list.

A=`1 → 4`, B=`2 → 3`. Links beyond tail may still point into an input until the next assignment replaces them.

Choice	Finalized result prefix	A head	B head
Initial	Empty	1	2
A: 1	1	4	2
B: 2	1 → 2	4	3
B: 3	1 → 2 → 3	4	None
Attach remaining A	1 → 2 → 3 → 4	4	None

21 · Turning the idea into code

```
from structures import ListNode

def merge_lists(first, second):
    dummy = tail = ListNode()
    while first and second:
        if first.val <= second.val:
            tail.next, first = first, first.next
        else:
            tail.next, second = second, second.next
        tail = tail.next
    tail.next = first or second
    return dummy.next
```

The dummy node makes the first attachment use the same operation as later attachments. Simultaneous assignment attaches the old first node and advances first to its original successor.

Once either list is empty, the other remainder is already sorted. `tail.next = first or second` attaches that chain in one operation. The returned head excludes dummy.

21 · Why it works; time and memory

The prefix through tail is sorted. The smaller head is minimal among all remaining nodes; the leftover suffix is already sorted.

For lengths n and m, each iteration finalizes one node. Alternating choices can require O(n+m) work. The final remainder attachment itself costs O(1).

One dummy node and a fixed number of references give O(1) auxiliary space. The result has n+m nodes, but these are reused input nodes. Preserving both input chains would require a copying implementation with O(n+m) new-node space.

Exercises and sources

LeetCode · 74, 33, 206, 21 · Worked solutions · Python ZIP

```
python demo_05.py
python demo_06.py
python verify.py
```

After implementing the student file, check it with `python verify.py --student`.

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