Week 1: Hashing and two pointers

Nearby duplicates in sensor logs

Retransmission produces the event log `[4, 7, 4, 4]`. We need to detect equal values at most one position apart. Comparing only the first and third 4 misses the final pair.

Repeatedly rescanning history becomes expensive as the log grows. This week retains only the past information needed by the question and uses sorted order to eliminate impossible candidates.

Week 1 problems and learning outcomes

Session 1: array, set, and dictionary costs; values linked to earlier positions. Session 2: justified pointer moves in sorted values; extension to three-value sums.

- Input, output, and constraint definitions
- Stored state and justification for discarding candidates
- Time, auxiliary storage, and output costs

Practice uses LeetCode 1, 219, 167, and 15.

Input size and one operation

n is the current number of elements. O(1) means the amount of work stays bounded as n grows; O(n) means the worst-case work grows proportionally to n. Constant complexity does not promise a smaller measured runtime for every input.

Task	n=4	n=8	Growth
Read one specified slot	1 slot	1 slot	Constant
Search to the end for an absent value	4 comparisons	8 comparisons	Proportional to n
Compare every distinct pair of positions	6 pairs	28 pairs	n(n-1)/2

We count bounded-integer and reference operations as O(1). Large integers and hashing or comparing long strings have additional length-dependent costs. Big-O describes a growth-rate upper bound; also specify whether the analysis is worst-case, expected, or amortized.

list: access by position versus search by value

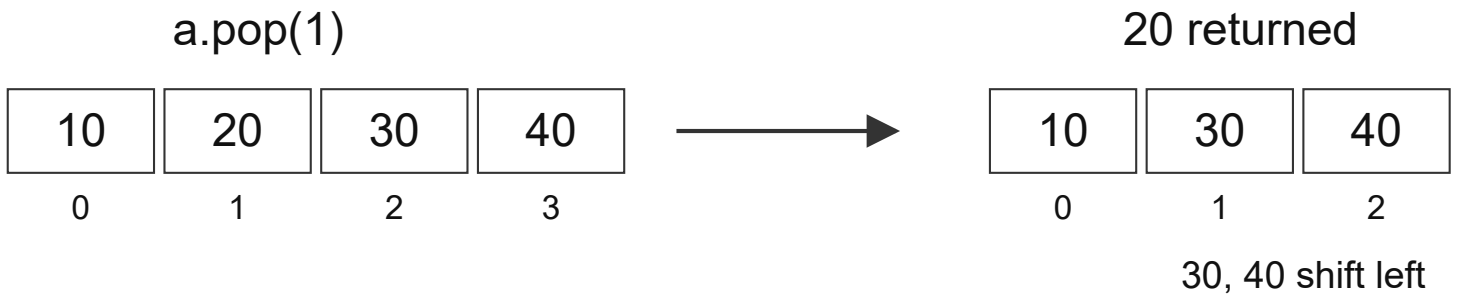
For `a=[10,20,30,40]`, `a[2]` directly accesses the third slot. CPython lists store references in an array, so the preceding values need not be compared. In contrast, `30 in a` does not know the position and compares values from the beginning.

Expression	Values examined	Result	Worst-case time
`a[2]`	Specified slot 2	30	O(1)
`a[2] = 99`	Specified slot 2	Replacement	O(1)
`30 in [10,20,30,40]`	10,20,30	True	O(n)
`50 in [10,20,30,40]`	10,20,30,40	False	O(n)
`a.index(40)`	Prefix through 40	Index 3	O(n)
`len(a)`	Stored length	4	O(1)

Knowing an index is different from searching for a value to discover its index. `.index` raises ValueError when the value is absent.

list: where removal happens

For `a=[10,20,30,40]`, `pop(1)` returns 20 and shifts references to 30 and 40 one slot left. Knowing the location does not eliminate the work of closing the gap.



Expression	Work performed	Time
`a.pop()`	Remove last; no trailing shifts	Amortized O(1)
`a.pop(i)`, `del a[i]`	Shift n-i-1 trailing references	O(n) worst at general positions
`a.remove(x)`	Find first x, then shift	O(n) worst case
`a.insert(0,x)`	Shift all references right	O(n)

The argument of pop is a position; the argument of remove is a value. `remove(20)` does not mean position two. Resizing costs are included in the amortized bounds.

append and amortized O(1)

A list reserves capacity beyond its current length. With spare capacity, append writes one reference at the end. When capacity is exhausted, a larger allocation may require moving the existing references, costing O(n) for that individual call.

Case	Work in one call	Aggregate cost
Append with spare capacity	Write one entry	O(1)
Append requiring growth	May move n references	O(n) for that call
N appends from an empty list	Spread growth costs across calls	O(N) total; O(1) amortized

Amortized analysis does not assume random input. It divides the total cost of an operation sequence by its length. It is different from guaranteeing constant time for every individual call.

Removing the last element is also amortized O(1) when resizing is included. This is the cost model used for list-based stacks in this course.

Sharing resize costs across calls

Count eight appends to an array whose capacity doubles through 1,2,4,8. This is a calculation for geometric growth, not CPython’s exact allocation schedule.

Length after append	New capacity needed	Existing references moved
1	1	0
2	2	1
3	4	2
4	unchanged	0
5	8	4
6–8	unchanged	0

Eight new writes plus 1+2+4=7 copies give 15 operations. When N is a power of two, copies total $N-1$ and all writes total $2N-1$. Between powers of two, copies remain below $2N$, so total cost is $O(N)$. Growing by just one slot would instead copy $1+2+\dots+(N-1)$ references, giving $O(N^2)$.

Dividing $O(N)$ total work by N calls gives amortized $O(1)$ per append.

Time and memory of copies and slices

```
a = [10, 20, 30, 40]
b = a
c = a[1:3]
a[1] = 99
print(b)  # [10, 99, 30, 40]
print(c)  # [20, 30]
```

`b=a` adds a reference to the same list, using O(1) time and auxiliary space. `a[1:3]` copies two references into a new list, so later slot replacement is independent.

Operation	Time	New storage
Slice q entries	O(q)	O(q)
`a.copy()`, `list(a)`	O(n)	O(n)
`sorted(a)`	O(n log n) worst case	O(n) copy/sorting workspace

With `rows=[[1],[2]]; copied=rows.copy()`, `copied[0].append(9)` changes both first rows to `[1,9]`: copied outer slots point to the same row object. Memory counts simultaneously retained storage, not the number of elements processed.

stack and queue: removal order

A stack is LIFO: last in, first out. A queue is FIFO: first in, first out. These describe access order. In Python, use the end of a list for a stack and the ends of a deque for a queue.

```
from collections import deque
stack = [10, 20, 30]
queue = deque([10, 20, 30])
print(stack.pop())     # 30
print(queue.popleft()) # 10
```

Purpose	Operation	Cost and reason
Stack insertion/removal	list.append / pop	End only; amortized O(1)
Queue insertion/removal	deque.append / popleft	O(1) at ends; no full shift
Insert at front	deque.appendleft	O(1)
Middle access/value search	q[i], x in q	O(n) worst case

A deque supports efficient end operations; arbitrary-position access is not constant-time like a list. The same queue costs apply to BFS in Weeks 4–6.

dict and set: information found by a key

A dict stores a value associated with each key; a set stores membership. Hashing narrows the locations to examine rather than comparing every element in order. Collisions can increase the number of comparisons.

```
latest = {4: 2, 7: 1}
print(4 in latest)          # True: tests keys
print(latest[4])            # 2: associated value
print(2 in latest.values()) # True: searches values
```

Operation	Expected/average time	Storage change
`key in d`, `d[key]`, `d.get(key)`	O(1)	No new entry
`d[key]=v`, `d.pop(key)`	O(1), growth amortized	Add/remove a key
`x in s`, `s.add(x)`, `s.discard(x)`	O(1), growth amortized	Add/remove an element
`v in d.values()`	O(n) worst case	Only a view added
`d.copy()`, `set(a)`	Expected O(n)	Up to O(n)

These bounds assume constant-cost hashing/equality and well-distributed keys. Severe collisions can make lookup O(n) in the worst case. Hashing a long string is not unconditionally O(1).

Add operation costs across the whole program

Removing every element can have different total costs depending on the operation. Repeated removal from the front of a list shifts n-1, n-2, …, 0 references.

Repeated operation	One operation	Count or total shifts	Total
list pop from end	Amortized O(1)	n calls	O(n)
list pop(0)	Current length	n(n-1)/2 shifts	O(n^2)
deque popleft	O(1)	n calls	O(n)
Search earlier list per value	Up to O(n)	n calls	O(n^2)
dict key lookup per value	Expected O(1)	n calls	Expected O(n)

Summing n values while retaining only a running sum takes O(n) time and O(1) auxiliary space. Count iterations and retained storage separately.

Input, auxiliary storage, and output

Implementation	Storage beyond input	Auxiliary space	Output space
Return only a sum	Sum and current value	O(1)	O(1)
Return prefix sums	Sum and current value	O(1)	O(n)
Store seen values in a set	d distinct values	O(d)	O(1) bool
Copy and sort n entries	Copy and sorting workspace	O(n)	Depends on return type
Recursion of depth h	h unfinished calls	O(h)	Depends on return type

Space measures the peak amount simultaneously retained. Overwriting a variable with each new value is different from continually appending to an output list.

Separate output from auxiliary space. A returned list with n entries and constant-size working variables means “O(1) auxiliary, O(n) output, O(n) combined.” Actual Python object byte sizes depend on implementation and value size.

CPython operation complexity · deque documentation

LeetCode 1: contract and baseline

Two Sum: return indices at distinct positions summing to target. The source guarantees one answer; one position cannot be reused.

For `nums=[4,8,3], target=7`, return `[0,2]`. Enumerating every pair needs no stored history.

```
def two_sum_brute(nums, target):
    for left in range(len(nums)):
        for right in range(left + 1, len(nums)):
            if nums[left] + nums[right] == target:
                return [left, right]
    return []
```

Pair counts and repeated searches

There are this many distinct index pairs. For n=10,000, the count is 49,995,000. An answer may be found earlier, but the worst case examines a quadratic number of candidates.

$$\frac{n\,(n-1)}{2}$$

If the current value is 3 and target is 7, the needed earlier value is 4. Store its presence and position instead of comparing every earlier value again.

Earlier-value state for Two Sum

`seen[value]` records an earlier position. Looking up before recording the current position ensures that the two returned indices differ.

```
def two_sum(nums, target):
    seen = {}
    for index, value in enumerate(nums):
        if target - value in seen:
            return [seen[target - value], index]
        seen[value] = index
    return []  # Defensive extension: official inputs have one answer.
```

Each iteration looks up the complement `target-value`. Store the current value only after an unsuccessful lookup, preventing reuse of the current position. With `[4,8,3]`, reading 3 finds 4:0 and recovers index 0.

index	value	Needed	seen before lookup	Result
0	4	3	{}	Absent → record 4:0
1	8	-1	{4:0}	Absent → record 8:1
2	3	4	{4:0,8:1}	Found → [0,2]

The map contains the distinct values already read, never future positions.

Two Sum correctness and update order

At loop entry, seen contains only earlier positions. When the later position of the answer is reached, the earlier partner is present.

For `nums=[3,3], target=6`, recording first can make lookup return the current index itself. The first 3 should fail lookup and be stored; the second then returns `[0,1]`.

Each element performs a fixed number of dictionary operations. Expected O(1) operations repeated n times give expected O(n) time. Up to n distinct value-position entries use O(n) auxiliary space; two returned indices use O(1). Severe hash collisions give an O(n^2) worst-case upper bound.

LeetCode 219: adding a distance condition

Contains Duplicate II: decide whether equal values at distinct positions are at most k apart.

`nums=[4,7,4,4], k=1` is True because of the final two entries. With k=0, no distinct indices qualify.

Rescanning up to k earlier positions per entry costs O(n·min(n,k)); slicing also copies each interval. Which single past occurrence per value is sufficient?

Meaning of the latest index

`latest[value]` is the most recent occurrence. It is the nearest past occurrence, so if it is too far, all older ones fail.

index	value	Prior latest[value]	Distance	Action
0	4	Absent	-	Store 4:0
1	7	Absent	-	Store 7:1
2	4	0	2 > 1	Fail, update to 4:2
3	4	2	1 <= 1	Return True

Implementing latest-index state

```
def nearby_duplicate(nums, k):
    latest = {}
    for index, value in enumerate(nums):
        if value in latest and index - latest[value] <= k:
            return True
        latest[value] = index
    return False
```

Update the latest position even when a repeated value fails the distance check. Keeping only the first occurrence misses the final pair in the example.

latest stores the most recent position for each value. Update it even after a failed distance check so the next occurrence uses its nearest predecessor. Only one position per distinct value is stored, although old keys are never deleted.

Correctness and storage of recent state

At current position j, the largest earlier matching index i minimizes j-i. If it is within k, an answer exists; otherwise every older occurrence fails. Updating preserves this meaning for the next iteration.

Check: explain why all previous indices of value 4 need not be stored.

One lookup, distance comparison, and update per value give expected O(n) time. There are d entries for d distinct values, so auxiliary space is O(d), d<=n. Even with k=1, all-distinct input leaves n keys; this implementation is not O(k)-space.

Session 1 concept checks

- What do `4 in events` and `4 in latest` each test?
- When does recording before lookup go wrong on `[3,3]`?
- Which answer is missed if index 2 does not update state in `[4,7,4,4], k=1`?
- How does copying latest on every iteration change complexity?

Check: membership tests list elements or dictionary keys. Storing first can return `[0,0]` on the first iteration of `[3,3]`. Skipping the latest-index update misses the last two 4s. Copying the dictionary each time can copy $0+1+\ldots+(n-1)$ entries, giving expected $O(n^2)$ time and peak $O(n)$ extra space even if only one copy is retained.

Pairing sorted part lengths

Find two part lengths summing to 7 in sorted `[1,4,6,9]`. The endpoints 1 and 9 total 10. Why is it safe to discard 9?

Session 2 uses order to eliminate many candidates at once. Before memorizing pointer moves, identify every pair removed by each move.

The input contract of LeetCode 167

Two Sum II - Input Array Is Sorted: input is nondecreasing and has one answer. Use distinct elements, return one-based positions, and use constant extra space.

```
numbers=[1,4,6,9], target=7 → [1,3]
```

This problem requires O(1) auxiliary space. Storing earlier values in a dictionary uses up to O(n) space, so we apply two pointers to the sorted input.

Eliminating a left endpoint with a small sum

The candidate interval is `[left,right]`. If its endpoint sum is below target, every available partner for left is at most the right value. All those sums are too small.

Incrementing left removes only failed pairs using that endpoint. Decreasing right cannot repair a sum that is too small.

Invariant: if an answer has not yet been found, both its positions remain inside the interval.

Eliminating a right endpoint with a large sum

If the endpoint sum exceeds target, every available partner for right is at least the left value. All those sums are too large.

Current pair	Sum	Decision for target=7	Removed candidates
1,9	10	Too large	9 paired with 1,4,6
1,6	7	Answer	No further elimination

The condition is left<right because one element cannot be used twice.

Implementing two pointers

```
def two_sum_sorted(numbers, target):
    left, right = 0, len(numbers) - 1
    while left < right:
        total = numbers[left] + numbers[right]
        if total == target:
            return [left + 1, right + 1]
        if total < target:
            left += 1
        else:
            right -= 1
    return []
```

Use zero-based indices inside the loop; add one only when returning.

When the sum is too small, every pair using the current left value is too small; advance left. When it is too large, discard the right value. Each move narrows the remaining candidates without revisiting discarded pairs.

Termination and total pointer movement

Each iteration returns an answer or moves one pointer inward. The gap shrinks, ensuring termination.

Outside the sorted contract, `[3,1,4], target=5` makes the rule miss 1 and 4. The elimination depends on the source problem’s sorting guarantee.

The initial gap right-left is n-1 and decreases by at least one per iteration, giving O(n) comparisons. Two pointers and a sum use O(1) auxiliary space; the two returned positions use O(1). No copy or sorting step is present.

LeetCode 15: three values and duplicate triplets

3Sum: return unique value triplets summing to zero at distinct positions. Equal values remain usable when their positions differ.

`[-2,0,1,1,2] → [[-2,0,2],[-2,1,1]]`

Enumerating three positions costs $O(n^3)$. Fixing one value leaves a two-value target equal to its negative.

Sorting and the fixed value

Sort a copy, fix index as the first position, and set left=index+1 and right=n-1. The three positions are then distinct.

Move left for a negative three-value sum and right for a positive one. This uses the same sorted elimination as 167.

3Sum returns value combinations, so we search the sorted array. Problems that return original indices require storing each value together with its original index.

Tracing 3Sum with fixed value -2

index,left,right	Values	Sum	Action
0,1,4	-2,0,2	0	Store [-2,0,2]; move both
0,2,3	-2,1,1	0	Store [-2,1,1]; move both
0,3,2	Interval ended	-	Finish fixed value -2

The next fixed value is 0. No pair from 1,1,2 sums to its negation, zero. Separate the search for one fixed value from subsequent fixed values.

3Sum implementation: fixing and moving

```
def three_sum(nums):
    values = sorted(nums)
    result = []
    for index in range(len(values) - 2):
        if index and values[index] == values[index - 1]:
            continue
        left, right = index + 1, len(values) - 1
        while left < right:
            total = values[index] + values[left] + values[right]
            if total < 0:
                left += 1
```

For each fixed value, restart two pointers in the suffix. Skip repeated fixed values and, after recording a match, repeated values at both ends to avoid duplicate value combinations. Equal values at distinct positions may still be used together.

3Sum implementation: recording and duplicate skipping

```
            elif total > 0:
                right -= 1
            else:
                result.append([values[index], values[left], values[right]])
                left += 1
                right -= 1
                while left < right and values[left] == values[left - 1]:
                    left += 1
                while left < right and values[right] == values[right + 1]:
                    right -= 1
    return result
```

After storing a match, move both pointers and skip values equal to the ones just used.

For `[-2,0,0,2,2]`, the first answer uses positions (0,1,4). Moving both pointers gives (left,right)=(2,3). Since `values[2]==values[1]==0`, left becomes 3 and the scan ends. The extra 0 and 2 would reproduce `[-2,0,2]`, so no new value combination is lost.

Two places to remove duplicate output

First, skip a fixed value equal to the previous fixed value. Second, after a match skip repeated left and right values. Equal values at distinct positions can be used together. Duplicate removal targets value combinations already returned.

For `[0,0,0,0]`, several index choices produce the single value triplet `[0,0,0]`. In `[-2,0,1,1,2]`, `[-2,1,1]` is valid because the two 1s occupy different positions.

3Sum correctness and cost

Every sorted triplet has a smallest value. When it is fixed, the two-pointer search includes its remaining pair and eliminates only impossible sums. Duplicate skipping removes repeated representations of the same value triplet.

There are at most n fixed positions, each with O(n) total pointer movement: O(n^2) time. Duplicate-skipping loops are part of those same movements, not another factor of n. O(n log n) sorting is dominated. Copy/sort workspace is O(n), separate from O(z) space for z output triplets.

Choosing state from the contract

Question	Retained state	Contract to check
Unsorted pair sum	Earlier value→position	No index reuse
Nearby equal values	Value→latest position	Distance k; update after failure
Sorted pair sum	left,right	Sorting, constant space, one-based output
Unique triplet sums	Fixed index and two pointers	Distinct indices, unique value triplets

Identify the information needed for the next decision and candidates that can be discarded before naming a data structure.

Independent exercises and submission

Solve LeetCode 1, 219, 167, and 15 in the student exercises. Implement student_week_01.py and trace a small input by hand first.

Submit state meaning, correctness, complexity, and boundary inputs with each solution. Use the staged hints and solutions only as far as needed.

Run `python week01_walkthrough.py` from the Python bundle to check the lecture examples and traces.

Core sources

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

Use each official LeetCode problem link for its complete input contract. Next week extends state updates to contiguous windows and unresolved information in stacks.