Week 1 Worked Companion

Operation costs and worked solutions

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.

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+…+(N−1) references, giving $O(N^2)$.

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

1 · Staged hints

Return two distinct indices whose values sum to target. An element cannot be reused; official inputs have one answer.

LeetCode 1: Two Sum · Easy

H1. Which value would complete the current value?

H2. When should an index be recorded to avoid using it twice?

H3. Store earlier value-to-index pairs; look up target-value before recording the current value.

`nums=[4,8,3], target=7`

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]

1 · Correctness and cost

At loop entry, the map contains only earlier positions. At the later position of the unique answer, its partner is present. This reuses information instead of checking all O(n^2) pairs.

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.

1 · Counterexamples and checks

Recording first can reuse one index on [3,3], target=6.

```
nums=[4,8,3], target=7 → [0,2]
```

1 · Implementation 1/1

```
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.

219 · Staged hints

Return True exactly when equal values occur at distinct positions i,j with distance $|i - j|$ at most k.

LeetCode 219: Contains Duplicate II · Easy

H1. Which occurrence of a value should be retained?

H2. Can an older occurrence be closer than the latest one?

H3. Retain the latest index per value; compare distance, then update.

`nums=[4,7,4,4], k=1`

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

219 · Correctness and cost

The latest occurrence minimizes distance to the current index. If it is too far, all older occurrences are too far. Updating preserves this property.

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.

219 · Counterexamples and checks

Keeping only the first position misses the last pair in [4,7,4,4], k=1. No distinct-index pair qualifies when k=0.

```
nums=[4,7,4,4], k=1 → True
```

219 · Implementation 1/1

```
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
```

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.

167 · Staged hints

Find two distinct positions summing to target in a nondecreasing array. Return 1-based indices using O(1) auxiliary space.

LeetCode 167: Two Sum II - Input Array Is Sorted · Medium

H1. If the endpoint sum is too small, which candidates are all too small?

H2. Does the elimination remain valid without sorting?

H3. While left<right, move left inward for a small sum and right inward for a large sum.

`nums=[1,4,6,9], target=7`

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

167 · Correctness and cost

If the sum is small, no partner at or below right works with left. If large, no partner at or above left works with right. Each elimination preserves any answer.

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.

167 · Counterexamples and checks

Outside the sorted contract, [3,1,4], target=5 defeats this rule. left==right reuses one element.

```
numbers=[1,4,6,9], target=10 → [1,4]
```

167 · Implementation 1/1

```
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 []
```

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.

15 · Staged hints

Return all distinct value triples from three distinct positions whose sum is zero. Results contain values rather than indices, with no repeated value combination.

LeetCode 15: 3Sum · Medium

H1. What remains after fixing one value?

H2. Does fixing the same value again create a new triplet?

H3. Sort a copy, fix the first value, and use two pointers on the suffix; skip repeated fixed values and matched pair values.

`nums=[-2,0,1,1,2]`

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

15 · Correctness and cost

Every ordered triplet is examined when its smallest value is fixed. Sorted elimination is valid; skipping duplicates removes repeated value combinations only.

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.

15 · Counterexamples and checks

[0,0,0,0] yields one triplet. Equal values at distinct positions remain allowed.

```
nums=[-2,0,1,1,2] → [[-2,0,2],[-2,1,1]]
```

15 · Implementation 1/2

```
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.

15 · Implementation 2/2

```
            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
```

When the sum is zero, record the triplet and move both ends inward. The following while loops skip repeated values so the same value combination is not emitted again. Both pointers keep moving inward, so these skips are part of the same linear scan.

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.

Execution and sources

```
cd code
python demo_01.py
python demo_02.py
python verify.py
```

Completed implementations are in solutions.py; independent work is in student_week_01.py. On LeetCode, adapt each function to the official method signature.

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