TwoSum Improvements and Analysis

CS 351: Analysis of Algorithms

Lucas P. Cordova, Ph.D.

Table of contents

1 Objectives

By the end of this lecture, you will:

- Understand the Zero-Sum Pairs counting problem

- Evaluate three different approaches with varying time complexities
- Understand hash tables and their performance benefits
- Analyze space-time tradeoffs in algorithm design
- Quantify performance improvements through complexity analysis
- Handle edge cases including duplicates and multiple pairs

2 Problem Statement

The Zero-Sum Pairs Challenge

Given an array of integers, count all pairs of elements that sum to exactly zero.
Requirements:

- Count all unique pairs (i, j) where i $<$ j
- arr[i] + arr[j] = 0
- Each element can be used in multiple pairs
- Return the total count of such pairs

2.1 Example Walkthrough

```python
from typing import List, Tuple, Dict, Optional

def example_input() -> Tuple[List[int], int]:
    """Demonstrate the problem with a simple example"""
    arr = [1, -1, 2, -2, 3, 0, 0]
    # Pairs that sum to 0:
    # (1, -1) at indices (0, 1)
    # (2, -2) at indices (2, 3)
    # (0, 0) at indices (5, 6)
    # Total: 3 pairs
    expected_count = 3
    return arr, expected_count

arr, expected = example_input()
print(f"Input array: {arr}")
print(f"Expected count: {expected}")
print("\nPairs that sum to zero:")
print("  arr[0] + arr[1] = 1 + (-1) = 0")
print("  arr[2] + arr[3] = 2 + (-2) = 0")
print("  arr[5] + arr[6] = 0 + 0 = 0")
```

```
Input array: [1, -1, 2, -2, 3, 0, 0]
Expected count: 3

Pairs that sum to zero:
  arr[0] + arr[1] = 1 + (-1) = 0
  arr[2] + arr[3] = 2 + (-2) = 0
  arr[5] + arr[6] = 0 + 0 = 0
```

2.2 Quick Check

1. Array[3, -3, 5, -5, 3], how many pairs sum to zero?
   - Answer: 2 pairs: (3, -3) and (5, -5)
2. Why only count pairs where i $<$ j?
   - Answer: To avoid double counting (i.e., (a,b) and (b,a))
3. Can the same element be part of multiple pairs? e.g. Array[1, -1, 1]
   - Answer: Yes! Example: In [1, -1, 1], the -1 at index 1 pairs with both 1s, forming 2 pairs total.
4. How is this different from finding pairs that sum to target = 10?
   - Answer: Same algorithm structure, but for zero-sum we can optimize by knowing we need x and -x (opposites). For arbitrary targets, we need x and (target-x).

3 Approach 1: Brute Force Solution

3.1 The Intuitive Approach

The most straightforward solution checks every possible pair of numbers.

```python
import time
import random
from typing import List, Tuple

def two_sum_brute_force(arr: List[int]) -> Tuple[int, int]:
    """
    Count pairs that sum to exactly 0 using brute force

    Args:
        arr: List of integers
```

```
    Returns:
        Tuple of (count of zero-sum pairs, number of comparisons made)

    Time Complexity: O(n²)
    Space Complexity: O(1)
    """
    count = 0
    n = len(arr)
    comparisons = 0

    for i in range(n):
        for j in range(i + 1, n):
            comparisons += 1
            if arr[i] + arr[j] == 0:
                count += 1

    return count, comparisons

# Test the brute force solution
test_arrays = [
    [1, -1, 2, -2, 3],
    [0, 0, 0],
    [5, -5, 10, -10, 5, -5],
    [1, 2, 3, 4, 5]
]

for arr in test_arrays:
    result, comps = two_sum_brute_force(arr)
    print(f"Array: {arr}")
    print(f"  Zero-sum pairs: {result}")
    print(f"  Comparisons made: {comps}")
    print()
```

```
Array: [1, -1, 2, -2, 3]
  Zero-sum pairs: 2
  Comparisons made: 10

Array: [0, 0, 0]
  Zero-sum pairs: 3
  Comparisons made: 3
```

```
Array: [5, -5, 10, -10, 5, -5]
  Zero-sum pairs: 5
  Comparisons made: 15

Array: [1, 2, 3, 4, 5]
  Zero-sum pairs: 0
  Comparisons made: 10
```

3.2 Visualizing Pair Detection

```python
def visualize_brute_force_pairs(arr: List[int]) -> None:
    """Visualize how brute force finds all zero-sum pairs"""

    print("Brute Force Pair Detection\n")
    print(f"Array: {arr}\n")

    n = len(arr)
    pairs_found = []
    comparison_count = 0

    print("Checking all pairs:")
    print("-" * 50)

    for i in range(n):
        for j in range(i + 1, n):
            comparison_count += 1
            sum_val = arr[i] + arr[j]
            is_zero = sum_val == 0

            status = "ZERO-SUM PAIR!" if is_zero else ""
            symbol = " " if is_zero else ""
            print(f"Compare [{i}]={arr[i]:3} + [{j}]={arr[j]:3} = {sum_val:3} {symbol} {statu

            if is_zero:
                pairs_found.append((i, j, arr[i], arr[j]))

    print("-" * 50)
    print(f"\nTotal comparisons: {comparison_count}")
    print(f"Zero-sum pairs found: {len(pairs_found)}")

    if pairs_found:
        print("\nPairs detail:")
```

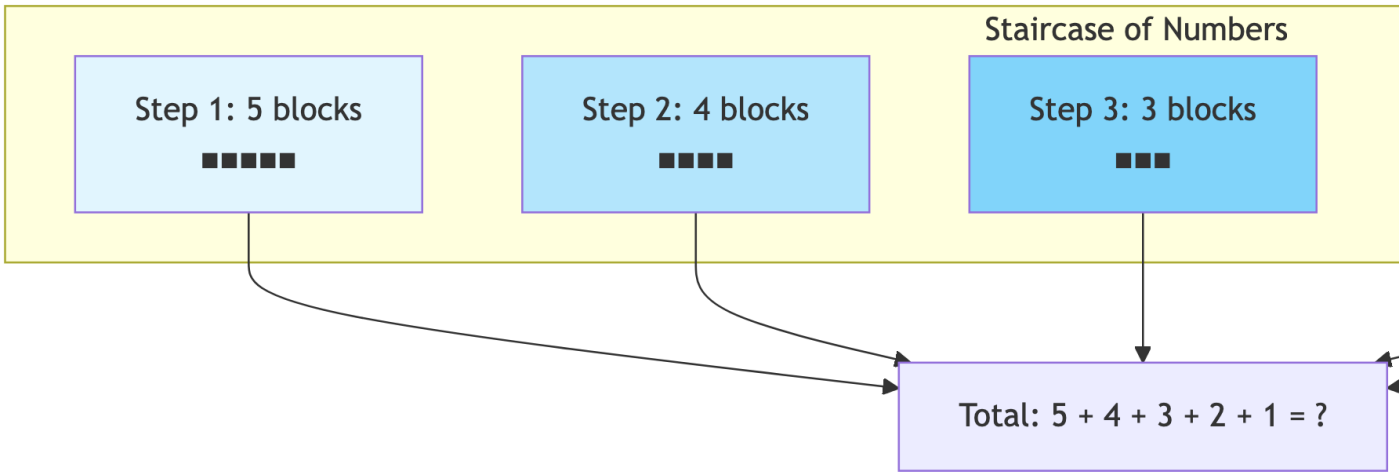
```python
        for i, j, val1, val2 in pairs_found:
            print(f"  indices ({i}, {j}): {val1} + {val2} = 0")

# Demonstrate with an example
demo_arr = [2, -2, 1, -1, 3, 0]
visualize_brute_force_pairs(demo_arr)
```

```
Brute Force Pair Detection

Array: [2, -2, 1, -1, 3, 0]

Checking all pairs:
--------------------------------------------------
Compare [0]=  2 + [1]= -2 =   0  ZERO-SUM PAIR!
Compare [0]=  2 + [2]=  1 =   3
Compare [0]=  2 + [3]= -1 =   1
Compare [0]=  2 + [4]=  3 =   5
Compare [0]=  2 + [5]=  0 =   2
Compare [1]= -2 + [2]=  1 =  -1
Compare [1]= -2 + [3]= -1 =  -3
Compare [1]= -2 + [4]=  3 =   1
Compare [1]= -2 + [5]=  0 =  -2
Compare [2]=  1 + [3]= -1 =   0  ZERO-SUM PAIR!
Compare [2]=  1 + [4]=  3 =   4
Compare [2]=  1 + [5]=  0 =   1
Compare [3]= -1 + [4]=  3 =   2
Compare [3]= -1 + [5]=  0 =  -1
Compare [4]=  3 + [5]=  0 =   3
--------------------------------------------------

Total comparisons: 15
Zero-sum pairs found: 2

Pairs detail:
  indices (0, 1): 2 + -2 = 0
  indices (2, 3): 1 + -1 = 0
```

4 Understanding Arithmetic Series (Simple Explanation)

4.1 What is Adding Numbers in a Pattern?

Imagine you have a staircase. The first step has 5 blocks, the second has 4 blocks, the third has 3 blocks, and so on. How many blocks total do you have?



This is called an **arithmetic series** - it's just a fancy name for adding up numbers that follow a pattern.

4.2 The Clever Trick to Add Them Quickly

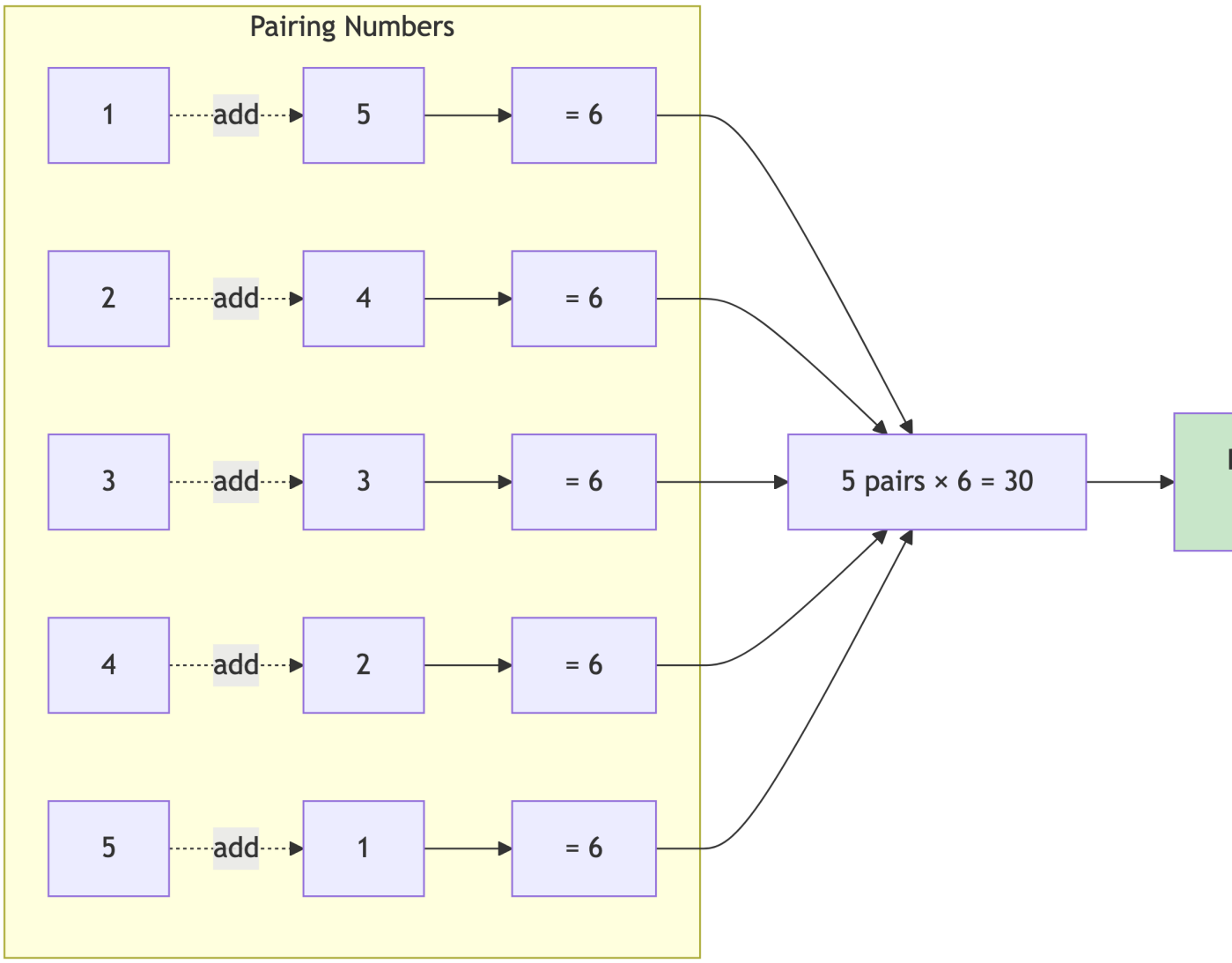
Here's a clever trick discovered by a mathematician named Gauss:

```
def gauss_trick_explanation() -> None:
    print("THE GAUSS TRICK: Adding 1 + 2 + 3 + 4 + 5")
    print("=" * 50)

    print("\nStep 1: Write the numbers forwards and backwards")
    print("  Forward:  1 + 2 + 3 + 4 + 5")
    print("  Backward: 5 + 4 + 3 + 2 + 1")

    print("\nStep 2: Add each column")
    print("  1 + 5 = 6")
    print("  2 + 4 = 6")
    print("  3 + 3 = 6")
    print("  4 + 2 = 6")
    print("  5 + 1 = 6")
```

```
15
16      print("\nStep 3: Notice the pattern!")
17      print("  We get 6 five times: 6 × 5 = 30")
18
19      print("\nStep 4: But wait! We counted everything twice")
20      print("  (once forward, once backward)")
21      print("  So the real answer is: 30 ÷ 2 = 15")
22
23      print("\n" + "=" * 50)
24      print("FORMULA: Sum = n × (n + 1) ÷ 2")
25      print("Where n is the last number you're adding to")
26      print("\nLet's verify: 5 × 6 ÷ 2 = 30 ÷ 2 = 15  ")
27
28  gauss_trick_explanation()
```

```
THE GAUSS TRICK: Adding 1 + 2 + 3 + 4 + 5
==================================================

Step 1: Write the numbers forwards and backwards
  Forward:  1 + 2 + 3 + 4 + 5
  Backward: 5 + 4 + 3 + 2 + 1

Step 2: Add each column
  1 + 5 = 6
  2 + 4 = 6
  3 + 3 = 6
  4 + 2 = 6
  5 + 1 = 6

Step 3: Notice the pattern!
  We get 6 five times: 6 × 5 = 30

Step 4: But wait! We counted everything twice
  (once forward, once backward)
  So the real answer is: 30 ÷ 2 = 15

==================================================
FORMULA: Sum = n × (n + 1) ÷ 2
Where n is the last number you're adding to

Let's verify: 5 × 6 ÷ 2 = 30 ÷ 2 = 15
```

4.3 Visual Representation of the Pairing Trick



4.4 How This Applies to Our Algorithm

```python
def explain_algorithm_arithmetic() -> None:
    """
    Connect arithmetic series to the brute force algorithm
    """
    print("HOW MANY PAIRS DO WE CHECK?")
    print("=" * 50)
```

```
7
8       print("\nImagine we have 5 numbers: [A, B, C, D, E]")
9       print("\nWe need to check these pairs:")
10      print("  From A: check with B, C, D, E → 4 pairs")
11      print("  From B: check with C, D, E    → 3 pairs")
12      print("  From C: check with D, E       → 2 pairs")
13      print("  From D: check with E          → 1 pair")
14      print("  From E: (no one left)         → 0 pairs")
15
16      print("\nTotal pairs: 4 + 3 + 2 + 1 + 0 = 10")
17
18      print("\n" + "-" * 50)
19      print("This is just like our staircase!")
20      print("We're adding: 4 + 3 + 2 + 1")
21
22      print("\nUsing the formula:")
23      print("  Sum = 4 × 5 ÷ 2 = 20 ÷ 2 = 10  ")
24
25      print("\nFor n items, we check:")
26      print("  (n-1) + (n-2) + ... + 2 + 1 pairs")
27      print("  Which equals: (n-1) × n ÷ 2")
28
29  explain_algorithm_arithmetic()
```

```
HOW MANY PAIRS DO WE CHECK?
==================================================

Imagine we have 5 numbers: [A, B, C, D, E]

We need to check these pairs:
  From A: check with B, C, D, E → 4 pairs
  From B: check with C, D, E    → 3 pairs
  From C: check with D, E       → 2 pairs
  From D: check with E          → 1 pair
  From E: (no one left)         → 0 pairs

Total pairs: 4 + 3 + 2 + 1 + 0 = 10

--------------------------------------------------
This is just like our staircase!
We're adding: 4 + 3 + 2 + 1
```

```
Using the formula:
  Sum = 4 × 5 ÷ 2 = 20 ÷ 2 = 10

For n items, we check:
  (n-1) + (n-2) + ... + 2 + 1 pairs
  Which equals: (n-1) × n ÷ 2
```

4.5 Deriving the Brute Force Complexity

```python
def analyze_brute_force_iterations(n: int) -> None:
    """
    Detailed analysis of how many comparisons the brute force algorithm makes
    """
    print(f"Analyzing Brute Force for array of size {n}\n")
    print("Nested Loop Structure:")
    print("for i in range(n):          # Outer loop")
    print("    for j in range(i+1, n):  # Inner loop")
    print("        compare(arr[i], arr[j])")
    print("\n" + "="*60)

    # Show iterations for each value of i
    print("\nIterations breakdown:")
    print(f"{'i':<10} {'j ranges from':<20} {'Comparisons':<15}")
    print("-"*45)

    total = 0
    iterations_per_i = []

    for i in range(n):
        j_start = i + 1
        j_end = n - 1
        count = n - (i + 1)
        iterations_per_i.append(count)
        total += count

        if i < 5 or i == n-1:  # Show first few and last
            if j_start <= j_end:
                print(f"{i:<10} {f'{j_start} to {j_end}':<20} {count:<15}")
            else:
                print(f"{i:<10} {'(none)':<20} {count:<15}")
        elif i == 5:
            print(f"{'...':<10} {'...':<20} {'...':<15}")
```

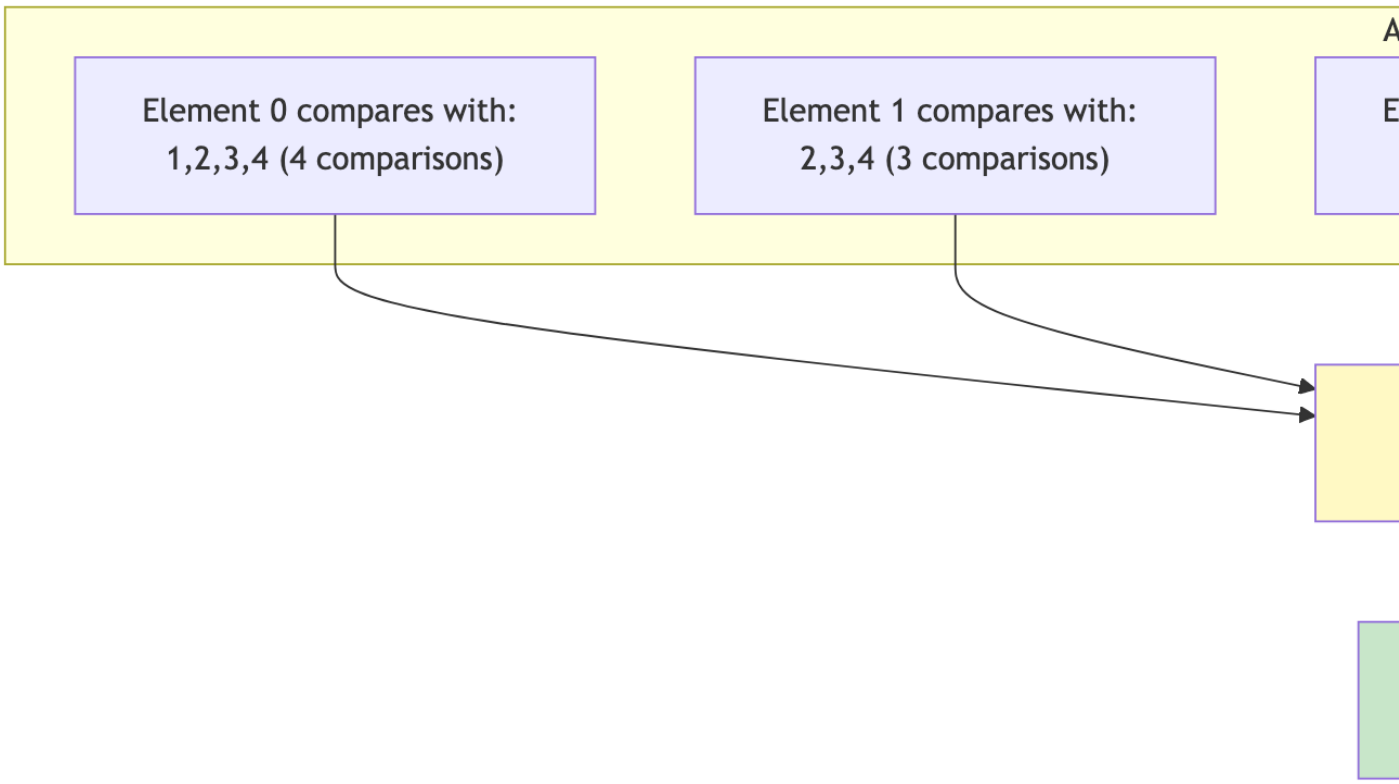
```
34
35      print("-"*45)
36      print(f"{'TOTAL':<10} {'':<20} {total:<15}")
37
38      # Show the arithmetic series
39      print("\n" + "="*60)
40      print("This forms an arithmetic series:")
41      series_str = " + ".join(str(x) for x in iterations_per_i[:min(5, n)])
42      if n > 5:
43          series_str += " + ... + " + str(iterations_per_i[-1])
44      print(f"Total = {series_str}")
45      print(f"Total = (n-1) + (n-2) + (n-3) + ... + 2 + 1")
46
47      # Derive the formula
48      print("\nMathematical Derivation:")
49      print("-"*40)
50      print("Let S = (n-1) + (n-2) + ... + 2 + 1")
51      print("This is the sum of first (n-1) natural numbers")
52      print("\nUsing the arithmetic series formula:")
53      print("Sum of 1 to k = k(k+1)/2")
54      print(f"\nTherefore: S = (n-1)((n-1)+1)/2")
55      print(f"           S = (n-1)(n)/2")
56      print(f"           S = {(n-1)*n//2}")
57
58      # Verify
59      calculated = (n-1) * n // 2
60      print(f"\nVerification: {total} = {calculated}  ")
61
62  # Demonstrate with different sizes
63  for size in [5, 10]:
64      analyze_brute_force_iterations(size)
65      print("\n" + "="*80 + "\n")
```

```
Analyzing Brute Force for array of size 5

Nested Loop Structure:
for i in range(n):          # Outer loop
    for j in range(i+1, n):  # Inner loop
        compare(arr[i], arr[j])

============================================================
```

```
Iterations breakdown:
i          j ranges from        Comparisons
-------------------------------------------
0          1 to 4               4
1          2 to 4               3
2          3 to 4               2
3          4 to 4               1
4          (none)               0
-------------------------------------------
TOTAL                           10

============================================================
This forms an arithmetic series:
Total = 4 + 3 + 2 + 1 + 0
Total = (n-1) + (n-2) + (n-3) + ... + 2 + 1

Mathematical Derivation:
---------------------------------------
Let S = (n-1) + (n-2) + ... + 2 + 1
This is the sum of first (n-1) natural numbers

Using the arithmetic series formula:
Sum of 1 to k = k(k+1)/2

Therefore: S = (n-1)((n-1)+1)/2
           S = (n-1)(n)/2
           S = 10

Verification: 10 = 10

================================================================================

Analyzing Brute Force for array of size 10

Nested Loop Structure:
for i in range(n):          # Outer loop
    for j in range(i+1, n):  # Inner loop
        compare(arr[i], arr[j])

============================================================

Iterations breakdown:
i          j ranges from        Comparisons
```

```
-------------------------------------------
0          1 to 9              9
1          2 to 9              8
2          3 to 9              7
3          4 to 9              6
4          5 to 9              5
...        ...                 ...
9          (none)              0
-------------------------------------------
TOTAL                          45

============================================================
This forms an arithmetic series:
Total = 9 + 8 + 7 + 6 + 5 + ... + 0
Total = (n-1) + (n-2) + (n-3) + ... + 2 + 1

Mathematical Derivation:
---------------------------------------
Let S = (n-1) + (n-2) + ... + 2 + 1
This is the sum of first (n-1) natural numbers

Using the arithmetic series formula:
Sum of 1 to k = k(k+1)/2

Therefore: S = (n-1)((n-1)+1)/2
           S = (n-1)(n)/2
           S = 45

Verification: 45 = 45

============================================================================================
```

4.6 Visualizing Loop Iterations as a Triangle

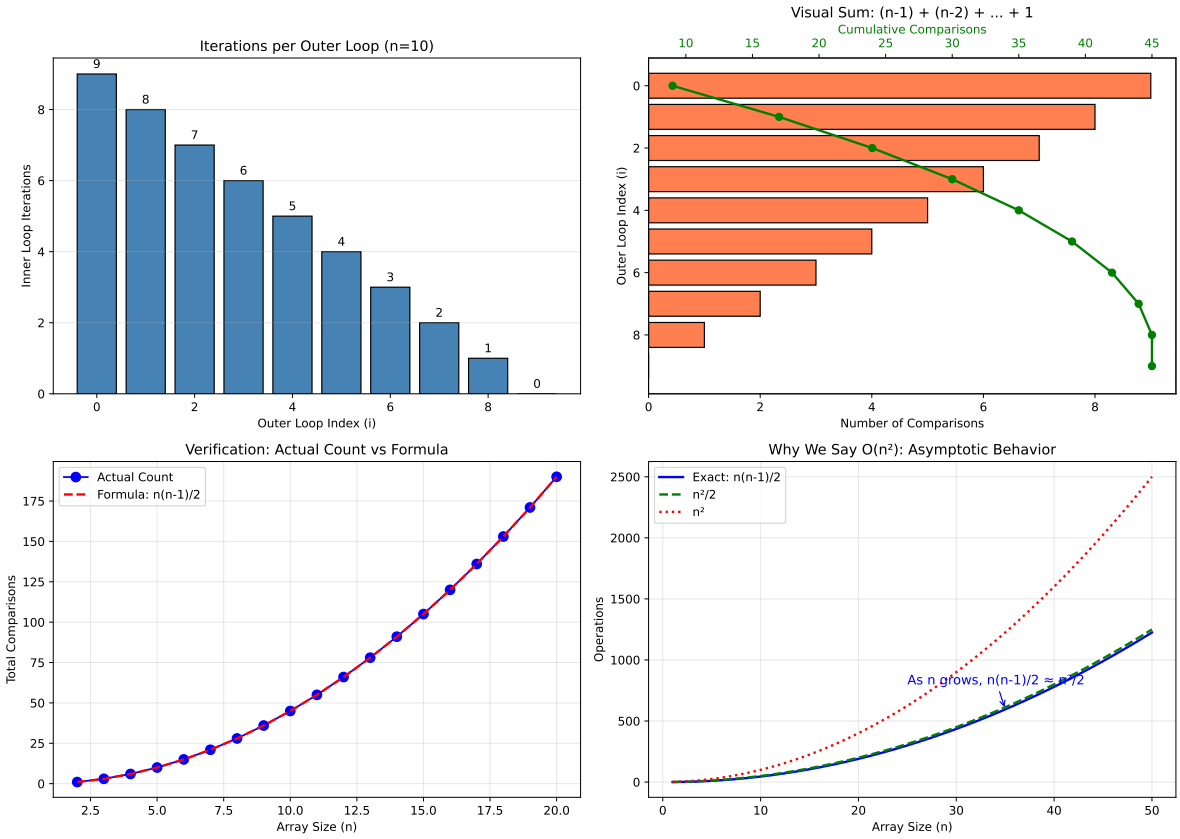


4.7 Visualizing the Arithmetic Series

```python
import matplotlib.pyplot as plt
import numpy as np

def visualize_arithmetic_series() -> None:
    """Visualize how the inner loop iterations form an arithmetic series"""

    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(14, 10))

    # Subplot 1: Bar chart of iterations per outer loop value
    n = 10
    iterations = [n - i - 1 for i in range(n)]

    ax1.bar(range(n), iterations, color='steelblue', edgecolor='black')
    ax1.set_xlabel('Outer Loop Index (i)')
```

```
    ax1.set_ylabel('Inner Loop Iterations')
    ax1.set_title(f'Iterations per Outer Loop (n={n})')
    ax1.grid(True, alpha=0.3, axis='y')

    # Add values on bars
    for i, val in enumerate(iterations):
        ax1.text(i, val + 0.1, str(val), ha='center', va='bottom')

    # Subplot 2: Visual representation of the series sum
    ax2.barh(range(n), iterations, color='coral', edgecolor='black')
    ax2.set_ylabel('Outer Loop Index (i)')
    ax2.set_xlabel('Number of Comparisons')
    ax2.set_title('Visual Sum: (n-1) + (n-2) + ... + 1')
    ax2.invert_yaxis()

    # Add cumulative sum line
    cumsum = np.cumsum(iterations)
    ax2_twin = ax2.twiny()
    ax2_twin.plot(cumsum, range(n), 'g-o', linewidth=2, markersize=6)
    ax2_twin.set_xlabel('Cumulative Comparisons', color='g')
    ax2_twin.tick_params(axis='x', labelcolor='g')

    # Subplot 3: Compare actual vs formula for different n
    sizes = range(2, 21)
    actual_counts = []
    formula_counts = []

    for size in sizes:
        actual = sum(size - i - 1 for i in range(size))
        formula = (size - 1) * size // 2
        actual_counts.append(actual)
        formula_counts.append(formula)

    ax3.plot(sizes, actual_counts, 'bo-', label='Actual Count', markersize=8)
    ax3.plot(sizes, formula_counts, 'r--', label='Formula: n(n-1)/2', linewidth=2)
    ax3.set_xlabel('Array Size (n)')
    ax3.set_ylabel('Total Comparisons')
    ax3.set_title('Verification: Actual Count vs Formula')
    ax3.legend()
    ax3.grid(True, alpha=0.3)

    # Subplot 4: Growth comparison - n² vs n(n-1)/2
    sizes = np.array(range(1, 51))
```

```
    exact = sizes * (sizes - 1) / 2
    n_squared = sizes ** 2
    n_squared_half = sizes ** 2 / 2

    ax4.plot(sizes, exact, 'b-', label='Exact: n(n-1)/2', linewidth=2)
    ax4.plot(sizes, n_squared_half, 'g--', label='n²/2', linewidth=2)
    ax4.plot(sizes, n_squared, 'r:', label='n²', linewidth=2)
    ax4.set_xlabel('Array Size (n)')
    ax4.set_ylabel('Operations')
    ax4.set_title('Why We Say O(n²): Asymptotic Behavior')
    ax4.legend()
    ax4.grid(True, alpha=0.3)

    # Add annotation
    ax4.annotate('As n grows, n(n-1)/2  n²/2',
                 xy=(35, 35*34/2), xytext=(25, 800),
                 arrowprops=dict(arrowstyle='->', color='blue'),
                 fontsize=11, color='blue')

    plt.tight_layout()
    plt.show()

visualize_arithmetic_series()
```



4.8 From Exact Count to Big-O Notation

```python
def explain_big_o_simplification(n: int = 1000) -> None:
    """
    Explain why n(n-1)/2 simplifies to O(n²)
    """
    print("From Exact Formula to Big-O Notation\n")
    print("="*60)

    exact = n * (n - 1) // 2

    print(f"For n = {n:,}:")
    print(f"  Exact comparisons: n(n-1)/2 = {exact:,}")
    print(f"  Expanded: n²/2 - n/2 = {n**2//2:,} - {n//2:,}")
    print(f"  n² term: {n**2:,}")
    print(f"  n²/2 term: {n**2//2:,}")

```

```
    print(f"\nRatio Analysis:")
    print(f"  exact / n² = {exact / n**2:.4f}")
    print(f"  exact / (n²/2) = {exact / (n**2/2):.4f}")

    print("\n" + "="*60)
    print("Big-O Simplification Rules:")
    print("-"*40)
    print("1. Drop constants: n²/2 → n²")
    print("2. Drop lower-order terms: n² - n → n²")
    print("3. Focus on dominant term as n → ∞")

    print(f"\nTherefore: n(n-1)/2 = O(n²)")

    # Show percentage contribution of each term
    print("\n" + "="*60)
    print("Term Contribution Analysis:")
    print("-"*40)

    for size in [10, 100, 1000, 10000]:
        n_squared_term = size ** 2 / 2
        n_term = size / 2
        total = n_squared_term - n_term

        n2_percent = (n_squared_term / total) * 100 if total > 0 else 0
        n_percent = abs(n_term / total) * 100 if total > 0 else 0

        print(f"n = {size:,}:")
        print(f"  n²/2 contributes: {n2_percent:.2f}%")
        print(f"  n/2 contributes:  {n_percent:.2f}%")
        print(f"  As n increases, n² term dominates!\n")

explain_big_o_simplification()
```

```
From Exact Formula to Big-O Notation

============================================================
For n = 1,000:
  Exact comparisons: n(n-1)/2 = 499,500
  Expanded: n²/2 - n/2 = 500,000 - 500
  n² term: 1,000,000
  n²/2 term: 500,000
```

```
Ratio Analysis:
  exact / n² = 0.4995
  exact / (n²/2) = 0.9990

============================================================
Big-O Simplification Rules:
----------------------------------------
1. Drop constants: n²/2 → n²
2. Drop lower-order terms: n² - n → n²
3. Focus on dominant term as n → ∞

Therefore: n(n-1)/2 = O(n²)

============================================================
Term Contribution Analysis:
----------------------------------------
n = 10:
  n²/2 contributes: 111.11%
  n/2 contributes:  11.11%
  As n increases, n² term dominates!

n = 100:
  n²/2 contributes: 101.01%
  n/2 contributes:  1.01%
  As n increases, n² term dominates!

n = 1,000:
  n²/2 contributes: 100.10%
  n/2 contributes:  0.10%
  As n increases, n² term dominates!

n = 10,000:
  n²/2 contributes: 100.01%
  n/2 contributes:  0.01%
  As n increases, n² term dominates!
```

4.9 Complexity Analysis

! Brute Force Performance - Complete Analysis

Exact Comparison Count: - First iteration (i=0): Compare with indices 1,2,...,n-1 = (n-1) comparisons - Second iteration (i=1): Compare with indices 2,3,...,n-1 = (n-2) comparisons - Third iteration (i=2): Compare with indices 3,4,...,n-1 = (n-3) comparisons - ... - Last iteration (i=n-2): Compare with index n-1 = 1 comparison - Final iteration (i=n-1): No comparisons = 0 comparisons
Total: (n-1) + (n-2) + (n-3) + ... + 2 + 1 + 0
Mathematical Derivation:

$$\text{Total} = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2}$$

Simplification to Big-O:

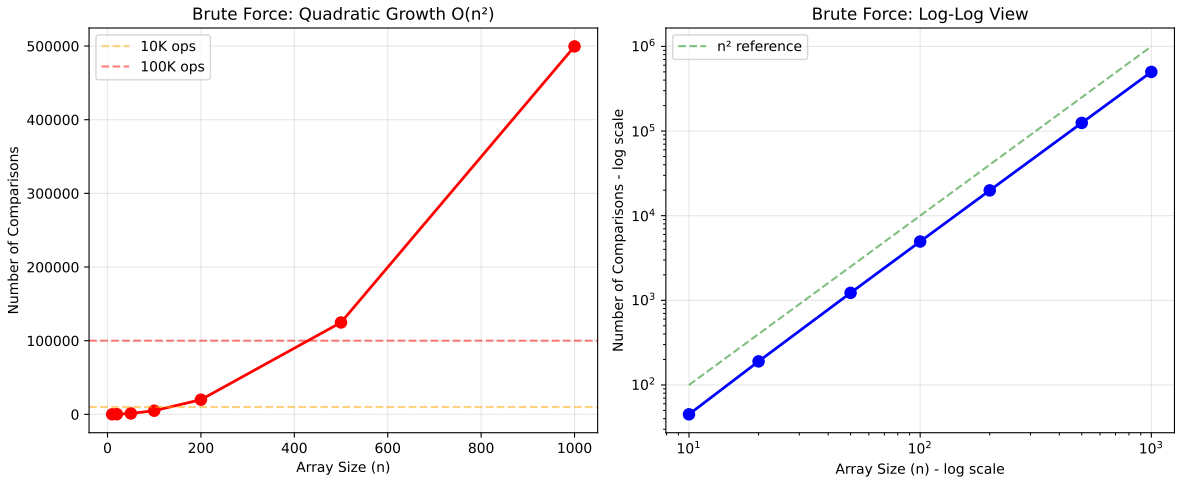
$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

As n grows large: - The $n^2/2$ term dominates - Constants are dropped in Big-O - Therefore: **$O(n^2)$**
Space Complexity: O(1) - Only using constant extra space for counters

4.10 Performance Scaling

```python
import matplotlib.pyplot as plt
import numpy as np

def visualize_brute_force_scaling() -> None:
    """Show how brute force performance degrades with array size"""

    sizes = [10, 20, 50, 100, 200, 500, 1000]
    comparisons = [(n * (n - 1)) // 2 for n in sizes]

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Linear scale
    ax1.plot(sizes, comparisons, 'ro-', linewidth=2, markersize=8)
    ax1.set_xlabel('Array Size (n)')
    ax1.set_ylabel('Number of Comparisons')
    ax1.set_title('Brute Force: Quadratic Growth O(n²)')
```

```
    ax1.grid(True, alpha=0.3)

    # Add reference lines
    ax1.axhline(y=10000, color='orange', linestyle='--', alpha=0.5, label='10K ops')
    ax1.axhline(y=100000, color='red', linestyle='--', alpha=0.5, label='100K ops')
    ax1.legend()

    # Log scale
    ax2.loglog(sizes, comparisons, 'bo-', linewidth=2, markersize=8)
    ax2.loglog(sizes, [n**2 for n in sizes], 'g--', alpha=0.5, label='n² reference')
    ax2.set_xlabel('Array Size (n) - log scale')
    ax2.set_ylabel('Number of Comparisons - log scale')
    ax2.set_title('Brute Force: Log-Log View')
    ax2.grid(True, alpha=0.3)
    ax2.legend()

    plt.tight_layout()
    plt.show()

visualize_brute_force_scaling()
```



4.11 Quick Check

1. Array size 6, how many comparisons?
    - Answer: 15 comparisons (5 + 4 + 3 + 2 + 1) or (n-1) * n/2 = 5 * 6/2 = 15
2. Why does inner loop start at i+1?

- Answer: To avoid double counting pairs and self-pairing (i.e., (a,b) and (b,a), or (a,a))

3. Double array size from 100 to 200, how many times more comparisons?
   - Answer: 4 times more comparisons. Since comparisons grow quadratically ($O(n^2)$), doubling n results in roughly 4 times the comparisons.
4. Using Gauss's formula for n=200, how many comparisons?
   - Answer: (n-1) * n/2 = (200-1) * 200/2 = 199 * 200/2 = 19900 comparisons
5. Why $O(n^2)$ instead of n(n-1)/2?
   - Answer: In Big-O notation, we drop constants and lower-order terms to focus on the dominant growth rate as n becomes large. Thus, n(n-1)/2 simplifies to $O(n^2)$.

5 Interlude: Understanding Hash Tables

5.1 What is a Hash Table?

Before diving into our optimized solution, let's understand the powerful data structure that makes it possible.

> **Hash Table Definition**
>
> A **hash table** (also called hash map or dictionary) is a data structure that implements an associative array - a structure that can map keys to values. It uses a **hash function** to compute an index into an array of buckets or slots, from which the desired value can be found.

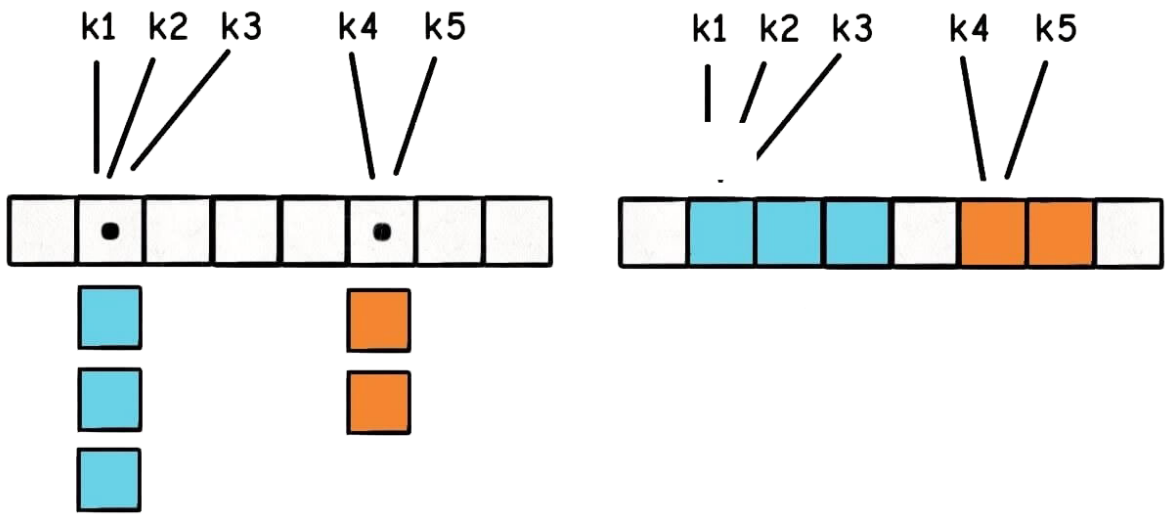


Figure 1: Figure: Hash Table Approaches (Chaining vs. Linear Probing)

5.2 Core Concepts

```python
from typing import Any, Optional

# In Python, dictionaries are implemented as hash tables
example_hash_table: Dict[str, int] = {}  # Empty hash table

# Adding key-value pairs - O(1) average time
example_hash_table["apple"] = 5
example_hash_table["banana"] = 3
example_hash_table["orange"] = 7

# Accessing values - O(1) average time
print(f"Value for 'apple': {example_hash_table['apple']}")

# Checking existence - O(1) average time
print(f"Is 'banana' in table? {'banana' in example_hash_table}")
print(f"Is 'grape' in table? {'grape' in example_hash_table}")

# For our zero-sum problem, we'll use hash tables to count occurrences
frequency_map: Dict[int, int] = {}
numbers = [1, -1, 2, -2, 1, -1, 3]

for num in numbers:
```

```
    frequency_map[num] = frequency_map.get(num, 0) + 1

print(f"\nFrequency map: {frequency_map}")
print(f"Count of 1: {frequency_map.get(1, 0)}")
print(f"Count of -1: {frequency_map.get(-1, 0)}")
```

```
Value for 'apple': 5
Is 'banana' in table? True
Is 'grape' in table? False

Frequency map: {1: 2, -1: 2, 2: 1, -2: 1, 3: 1}
Count of 1: 2
Count of -1: 2
```

5.3 How Hash Tables Work

```
def demonstrate_hash_function() -> None:
    """Demonstrate how hash functions map keys to array indices"""

    print("Hash Function Example for Integers\n")

    # Simple hash function for demonstration
    def simple_hash(key: int, table_size: int = 10) -> int:
        """A simple hash function for integers"""
        return abs(key) % table_size

    # Create a simple hash table representation
    table_size = 10
    hash_table: List[Optional[Tuple[int, int]]] = [None] * table_size

    numbers = [5, -5, 12, -12, 3, -3, 20, -20, 7, -7, 0, 4]

    print(f"Table size: {table_size} buckets\n")
    print("Hashing integers (for counting occurrences):")
    print("-" * 50)

    for num in numbers:
        index = simple_hash(num, table_size)
        print(f"Number: {num:3} → Hash index: {index}")

    print("\n" + "=" * 50)
    print("Important for Zero-Sum pairs:")
```

```
    print("- We need to quickly check if -x exists for each x")
    print("- Hash table gives us O(1) lookup time")
    print("- Perfect for counting pairs efficiently!")

demonstrate_hash_function()
```

```
Hash Function Example for Integers

Table size: 10 buckets

Hashing integers (for counting occurrences):
--------------------------------------------------
Number:   5 → Hash index: 5
Number:  -5 → Hash index: 5
Number:  12 → Hash index: 2
Number: -12 → Hash index: 2
Number:   3 → Hash index: 3
Number:  -3 → Hash index: 3
Number:  20 → Hash index: 0
Number: -20 → Hash index: 0
Number:   7 → Hash index: 7
Number:  -7 → Hash index: 7
Number:   0 → Hash index: 0
Number:   4 → Hash index: 4

==================================================
Important for Zero-Sum pairs:
- We need to quickly check if -x exists for each x
- Hash table gives us O(1) lookup time
- Perfect for counting pairs efficiently!
```

5.4 Why Hash Tables are Fast for Zero-Sum

```
import matplotlib.pyplot as plt
import numpy as np

def visualize_lookup_comparison() -> None:
    """Compare array search vs hash table lookup for finding pairs"""

    fig, ax = plt.subplots(1, 1, figsize=(10, 6))

    sizes = np.array([10, 100, 1000, 10000])
```

```
10
11      # For each element, searching for its negative
12      array_search = sizes * sizes / 2  # O(n²) total
13      hash_search = sizes  # O(n) total
14
15      ax.loglog(sizes, array_search, 'r-o', label='Nested Loop Search O(n²)', linewidth=2)
16      ax.loglog(sizes, hash_search, 'g-o', label='Hash Table Lookup O(n)', linewidth=2)
17
18      # Add speedup annotations
19      for i, size in enumerate(sizes):
20          speedup = array_search[i] / hash_search[i]
21          ax.annotate(f'{speedup:.0f}x faster',
22                      xy=(size, hash_search[i]),
23                      xytext=(size*1.2, hash_search[i]*2),
24                      fontsize=10, color='green')
25
26      ax.set_xlabel('Array Size (n)')
27      ax.set_ylabel('Total Operations')
28      ax.set_title('Finding Zero-Sum Pairs: Array Search vs Hash Table')
29      ax.legend()
30      ax.grid(True, alpha=0.3)
31
32      plt.tight_layout()
33      plt.show()
34
35  visualize_lookup_comparison()
```

5.5 Key Advantages for Zero-Sum Counting

Why Hash Tables Excel at Zero-Sum Pairs

1. **Instant Complement Check:** For each number x, check if -x exists in O(1)
2. **Frequency Counting:** Track how many times each number appears
3. **Handle Duplicates:** Count multiple occurrences efficiently
4. **Single Pass Possible:** Can solve in one iteration through array

Compare finding pairs:

- **Brute force:** Check all n(n-1)/2 pairs → $O(n^2)$
- **Hash table:** Check each element once → O(n)

5.6 Quick Check

1. Average time complexity for hash table lookup?
    - Answer: O(1) - constant time. This is possible because the hash function directly computes the memory location, no searching needed.

2. Example of hash collision:
    - Answer: If our hash function is (number % 10), then 15 and 25 would both hash to 5, causing a collision.
3. Memory for 1000 items vs 3 variables:
    - Answer: Roughly 333x more memory. Hash table needs to store both keys and values, plus overhead for the data structure itself.
4. What to store for zero-sum problem?
    - Answer: Store each unique number as key and its frequency (count) as value. Example: {2: 3, -2: 2} means we have three 2s and two -2s.
5. Why is checking "-x exists" fast in hash table but slow in array?
    - Answer: Hash table: O(1) direct lookup. Array: O(n) must scan through all elements sequentially to find if -x exists.

6 Approach 2: Hash Table Solution

6.1 The Optimized Approach

Using a hash table to count occurrences and find pairs efficiently.

```
from typing import List, Tuple, Dict

def two_sum_hash_table(arr: List[int]) -> Tuple[int, int]:
    """
    Count pairs that sum to exactly 0 using hash table

    Args:
        arr: List of integers

    Returns:
        Tuple of (count of zero-sum pairs, number of lookups made)

    Time Complexity: O(n)
    Space Complexity: O(n)
    """
    frequency: Dict[int, int] = {}
    lookups = 0

    # First pass: count frequencies
```

```
    for num in arr:
        frequency[num] = frequency.get(num, 0) + 1
        lookups += 1

    count = 0
    processed: set = set()

    # Second pass: count pairs
    for num in frequency:
        lookups += 1

        if num in processed:
            continue

        if num == 0:
            # Special case: pairs of zeros
            # Choose 2 from frequency[0] zeros: C(n,2) = n*(n-1)/2
            zeros = frequency[0]
            count += (zeros * (zeros - 1)) // 2
        elif -num in frequency:
            # Regular case: positive-negative pairs
            # Each positive can pair with each negative
            count += frequency[num] * frequency[-num]
            processed.add(num)
            processed.add(-num)

    return count, lookups

# Test the hash table solution
test_arrays = [
    [1, -1, 2, -2, 3],
    [0, 0, 0],
    [5, -5, 10, -10, 5, -5],
    [1, 2, 3, 4, 5]
]

print("Hash Table Approach Results:")
print("-" * 50)
for arr in test_arrays:
    result, lookups = two_sum_hash_table(arr)
    print(f"Array: {arr}")
    print(f"  Zero-sum pairs: {result}")
    print(f"  Hash operations: {lookups}")
```

```
    print()

Hash Table Approach Results:
--------------------------------------------------
Array: [1, -1, 2, -2, 3]
  Zero-sum pairs: 2
  Hash operations: 10

Array: [0, 0, 0]
  Zero-sum pairs: 3
  Hash operations: 4

Array: [5, -5, 10, -10, 5, -5]
  Zero-sum pairs: 5
  Hash operations: 10

Array: [1, 2, 3, 4, 5]
  Zero-sum pairs: 0
  Hash operations: 10
```

6.2 Step-by-Step Visualization

```python
def demonstrate_hash_table_counting(arr: List[int]) -> None:
    """Visualize how hash table approach counts zero-sum pairs"""

    print("Hash Table Approach - Step by Step\n")
    print(f"Input array: {arr}\n")

    # Step 1: Build frequency map
    print("STEP 1: Build Frequency Map")
    print("-" * 40)

    frequency: Dict[int, int] = {}
    for i, num in enumerate(arr):
        frequency[num] = frequency.get(num, 0) + 1
        print(f"  Process arr[{i}] = {num:3} → frequency = {dict(frequency)}")

    print(f"\nFinal frequency map: {frequency}\n")

    # Step 2: Count pairs
    print("STEP 2: Count Zero-Sum Pairs")
    print("-" * 40)
```

```python

    count = 0
    processed: set = set()

    for num in sorted(frequency.keys()):
        if num in processed:
            continue

        print(f"\nChecking number: {num}")

        if num == 0:
            zeros = frequency[0]
            pairs_from_zeros = (zeros * (zeros - 1)) // 2
            print(f"  Special case: {zeros} zeros")
            print(f"  Can form C({zeros},2) = {pairs_from_zeros} pairs")
            count += pairs_from_zeros
        elif -num in frequency:
            pairs = frequency[num] * frequency[-num]
            print(f"  Found complement: {-num}")
            print(f"  {frequency[num]} instances of {num} × {frequency[-num]} instances of {
            print(f"  Forms {pairs} pairs")
            count += pairs
            processed.add(num)
            processed.add(-num)
        else:
            print(f"  No complement ({-num}) found")

    print("\n" + "=" * 40)
    print(f"TOTAL ZERO-SUM PAIRS: {count}")

# Demonstrate with examples
demo_arrays = [
    [1, -1, 2, -2, 1, -1],
    [0, 0, 0, 1, -1],
    [5, -5, 5, -5, 10]
]

for arr in demo_arrays:
    demonstrate_hash_table_counting(arr)
    print("\n" + "=" * 60 + "\n")
```

Hash Table Approach - Step by Step

```
Input array: [1, -1, 2, -2, 1, -1]

STEP 1: Build Frequency Map
----------------------------------------
  Process arr[0] =   1 → frequency = {1: 1}
  Process arr[1] =  -1 → frequency = {1: 1, -1: 1}
  Process arr[2] =   2 → frequency = {1: 1, -1: 1, 2: 1}
  Process arr[3] =  -2 → frequency = {1: 1, -1: 1, 2: 1, -2: 1}
  Process arr[4] =   1 → frequency = {1: 2, -1: 1, 2: 1, -2: 1}
  Process arr[5] =  -1 → frequency = {1: 2, -1: 2, 2: 1, -2: 1}

Final frequency map: {1: 2, -1: 2, 2: 1, -2: 1}

STEP 2: Count Zero-Sum Pairs
----------------------------------------

Checking number: -2
  Found complement: 2
  1 instances of -2 × 1 instances of 2
  Forms 1 pairs

Checking number: -1
  Found complement: 1
  2 instances of -1 × 2 instances of 1
  Forms 4 pairs

========================================
TOTAL ZERO-SUM PAIRS: 5

============================================================

Hash Table Approach - Step by Step

Input array: [0, 0, 0, 1, -1]

STEP 1: Build Frequency Map
----------------------------------------
  Process arr[0] =   0 → frequency = {0: 1}
  Process arr[1] =   0 → frequency = {0: 2}
  Process arr[2] =   0 → frequency = {0: 3}
  Process arr[3] =   1 → frequency = {0: 3, 1: 1}
  Process arr[4] =  -1 → frequency = {0: 3, 1: 1, -1: 1}
```

```
Final frequency map: {0: 3, 1: 1, -1: 1}

STEP 2: Count Zero-Sum Pairs
----------------------------------------

Checking number: -1
  Found complement: 1
  1 instances of -1 × 1 instances of 1
  Forms 1 pairs

Checking number: 0
  Special case: 3 zeros
  Can form C(3,2) = 3 pairs

========================================
TOTAL ZERO-SUM PAIRS: 4

============================================================

Hash Table Approach - Step by Step

Input array: [5, -5, 5, -5, 10]

STEP 1: Build Frequency Map
----------------------------------------
  Process arr[0] =   5 → frequency = {5: 1}
  Process arr[1] =  -5 → frequency = {5: 1, -5: 1}
  Process arr[2] =   5 → frequency = {5: 2, -5: 1}
  Process arr[3] =  -5 → frequency = {5: 2, -5: 2}
  Process arr[4] =  10 → frequency = {5: 2, -5: 2, 10: 1}

Final frequency map: {5: 2, -5: 2, 10: 1}

STEP 2: Count Zero-Sum Pairs
----------------------------------------

Checking number: -5
  Found complement: 5
  2 instances of -5 × 2 instances of 5
  Forms 4 pairs

Checking number: 10
```

```
  No complement (-10) found

========================================
TOTAL ZERO-SUM PAIRS: 4

============================================================
```

6.3 Handling Edge Cases

```python
def two_sum_hash_table_detailed(arr: List[int]) -> int:
    """
    Enhanced version with edge case handling

    Time Complexity: O(n)
    Space Complexity: O(n)
    """
    if not arr or len(arr) < 2:
        return 0

    frequency: Dict[int, int] = {}

    # Count frequencies
    for num in arr:
        frequency[num] = frequency.get(num, 0) + 1

    count = 0

    # Handle zeros separately (pairs within same value)
    if 0 in frequency:
        zeros = frequency[0]
        count += (zeros * (zeros - 1)) // 2

    # Handle positive-negative pairs
    seen: set = set()
    for num in frequency:
        if num > 0 and -num in frequency and num not in seen:
            count += frequency[num] * frequency[-num]
            seen.add(num)
            seen.add(-num)

    return count

```

```python
# Test edge cases
edge_cases = [
    [],                    # Empty array
    [1],                   # Single element
    [0, 0, 0, 0],          # All zeros
    [1, 1, 1, 1],          # All same (non-zero)
    [1, -2, 3, -4, 5],     # No pairs
    [100, -100] * 50       # Many duplicates
]

print("Edge Case Testing:")
print("-" * 50)
for arr in edge_cases:
    result = two_sum_hash_table_detailed(arr)
    display_arr = arr if len(arr) <= 10 else f"{arr[:5]} ... {arr[-5:]}"
    print(f"Array: {display_arr}")
    print(f"  Zero-sum pairs: {result}\n")
```

```
Edge Case Testing:
--------------------------------------------------
Array: []
  Zero-sum pairs: 0

Array: [1]
  Zero-sum pairs: 0

Array: [0, 0, 0, 0]
  Zero-sum pairs: 6

Array: [1, 1, 1, 1]
  Zero-sum pairs: 0

Array: [1, -2, 3, -4, 5]
  Zero-sum pairs: 0

Array: [100, -100, 100, -100, 100] ... [-100, 100, -100, 100, -100]
  Zero-sum pairs: 2500
```

6.4 Quick Check

1. Main tradeoff between brute force and hash table?

- Answer: Time vs Space. Brute force uses O(1) space but $O(n^2)$ time. Hash table uses O(n) space but only O(n) time.

2. Single pass hash table possible?
   - Answer: Yes! As we iterate, check if -current exists in map, count pairs, then add current to map. But the two-pass version is clearer and handles duplicates more easily.

7 Approach 3: Sorted Array with Two Pointers

7.1 Alternative Approach for Sorted Data

If we can sort the array, we can use two pointers to find pairs efficiently.

```python
from typing import List, Tuple

def two_sum_two_pointer(arr: List[int]) -> Tuple[int, int]:
    """
    Count pairs that sum to exactly 0 using two pointers

    Args:
        arr: List of integers

    Returns:
        Tuple of (count of zero-sum pairs, number of comparisons)

    Time Complexity: O(n log n) due to sorting
    Space Complexity: O(1) excluding sort space
    """
    if len(arr) < 2:
        return 0, 0

    # Sort the array
    sorted_arr = sorted(arr)
    n = len(sorted_arr)

    left = 0
    right = n - 1
    count = 0
    comparisons = 0

```

```
    while left < right:
        comparisons += 1
        current_sum = sorted_arr[left] + sorted_arr[right]

        if current_sum == 0:
            # Found a zero-sum pair
            # Count duplicates on both sides
            left_val = sorted_arr[left]
            right_val = sorted_arr[right]

            if left_val == right_val:
                # Special case: all remaining elements are the same (zeros)
                remaining = right - left + 1
                count += (remaining * (remaining - 1)) // 2
                break

            # Count duplicates
            left_count = 1
            right_count = 1

            # Count duplicates on the left
            while left + left_count < right and sorted_arr[left + left_count] == left_val:
                left_count += 1

            # Count duplicates on the right
            while right - right_count > left and sorted_arr[right - right_count] == right_val
                right_count += 1

            # Each left element pairs with each right element
            count += left_count * right_count

            # Move pointers past duplicates
            left += left_count
            right -= right_count

        elif current_sum < 0:
            left += 1
        else:
            right -= 1

    return count, comparisons

# Test the two-pointer solution
```

```python
test_arrays = [
    [1, -1, 2, -2, 3],
    [0, 0, 0],
    [5, -5, 10, -10, 5, -5],
    [1, 2, 3, 4, 5]
]

print("Two-Pointer Approach Results:")
print("-" * 50)
for arr in test_arrays:
    result, comps = two_sum_two_pointer(arr)
    print(f"Array: {arr}")
    print(f"  Sorted: {sorted(arr)}")
    print(f"  Zero-sum pairs: {result}")
    print(f"  Comparisons: {comps}")
    print()
```

```
Two-Pointer Approach Results:
--------------------------------------------------
Array: [1, -1, 2, -2, 3]
  Sorted: [-2, -1, 1, 2, 3]
  Zero-sum pairs: 2
  Comparisons: 3

Array: [0, 0, 0]
  Sorted: [0, 0, 0]
  Zero-sum pairs: 3
  Comparisons: 1

Array: [5, -5, 10, -10, 5, -5]
  Sorted: [-10, -5, -5, 5, 5, 10]
  Zero-sum pairs: 5
  Comparisons: 2

Array: [1, 2, 3, 4, 5]
  Sorted: [1, 2, 3, 4, 5]
  Zero-sum pairs: 0
  Comparisons: 4
```

7.2 Visualizing Two-Pointer Movement

```python
def visualize_two_pointer_process(arr: List[int]) -> None:
    """Visualize how two pointers find zero-sum pairs"""

    sorted_arr = sorted(arr)
    n = len(sorted_arr)

    print("Two-Pointer Technique Visualization\n")
    print(f"Original: {arr}")
    print(f"Sorted:   {sorted_arr}\n")

    left = 0
    right = n - 1
    step = 1
    total_pairs = 0

    while left < right:
        # Visual representation
        visual = [' ' * 4] * n
        visual[left] = ' L  '
        visual[right] = '  R '

        print(f"Step {step}:")
        print("  " + "".join(f"[{num:3}]" for num in sorted_arr))
        print("  " + "".join(visual))

        current_sum = sorted_arr[left] + sorted_arr[right]
        print(f"  Sum: {sorted_arr[left]:3} + {sorted_arr[right]:3} = {current_sum:3}", end="

        if current_sum == 0:
            # Count duplicates for accurate pair counting
            left_val, right_val = sorted_arr[left], sorted_arr[right]
            left_count = 1
            right_count = 1

            while left + left_count < right and sorted_arr[left + left_count] == left_val:
                left_count += 1
            while right - right_count > left and sorted_arr[right - right_count] == right_val
                right_count += 1

            pairs = left_count * right_count
```

```
            total_pairs += pairs
            print(f"Found {pairs} pair(s)!")

            left += left_count
            right -= right_count
        elif current_sum < 0:
            print("→ Move left pointer")
            left += 1
        else:
            print("← Move right pointer")
            right -= 1

        step += 1
        print()

    print(f"Total zero-sum pairs found: {total_pairs}")

# Demonstrate with example
demo_arr = [3, -3, 1, -1, 2, -2, 0, 0]
visualize_two_pointer_process(demo_arr)
```

```
Two-Pointer Technique Visualization

Original: [3, -3, 1, -1, 2, -2, 0, 0]
Sorted:   [-3, -2, -1, 0, 0, 1, 2, 3]

Step 1:
  [ -3][ -2][ -1][  0][  0][  1][  2][  3]
   L                             R
  Sum:  -3 +   3 =   0 Found 1 pair(s)!

Step 2:
  [ -3][ -2][ -1][  0][  0][  1][  2][  3]
       L                     R
  Sum:  -2 +   2 =   0 Found 1 pair(s)!

Step 3:
  [ -3][ -2][ -1][  0][  0][  1][  2][  3]
            L            R
  Sum:  -1 +   1 =   0 Found 1 pair(s)!

Step 4:
```

```
  [ -3][ -2][ -1][  0][  0][  1][  2][  3]
               L    R
  Sum:   0 +   0 =   0 Found 1 pair(s)!

Total zero-sum pairs found: 4
```

7.3 Quick Check

1. Why must array be sorted?
    - Answer: The algorithm relies on the property that if sum is too small, we need a larger number (move left pointer right), and if sum is too large, we need a smaller number (move right pointer left). This only works with sorted order.
2. Time complexity of sorting?
    - Answer: O(n log n) for efficient sorting algorithms. This dominates the O(n) scanning phase, so overall complexity is O(n log n).

8 Performance Comparison

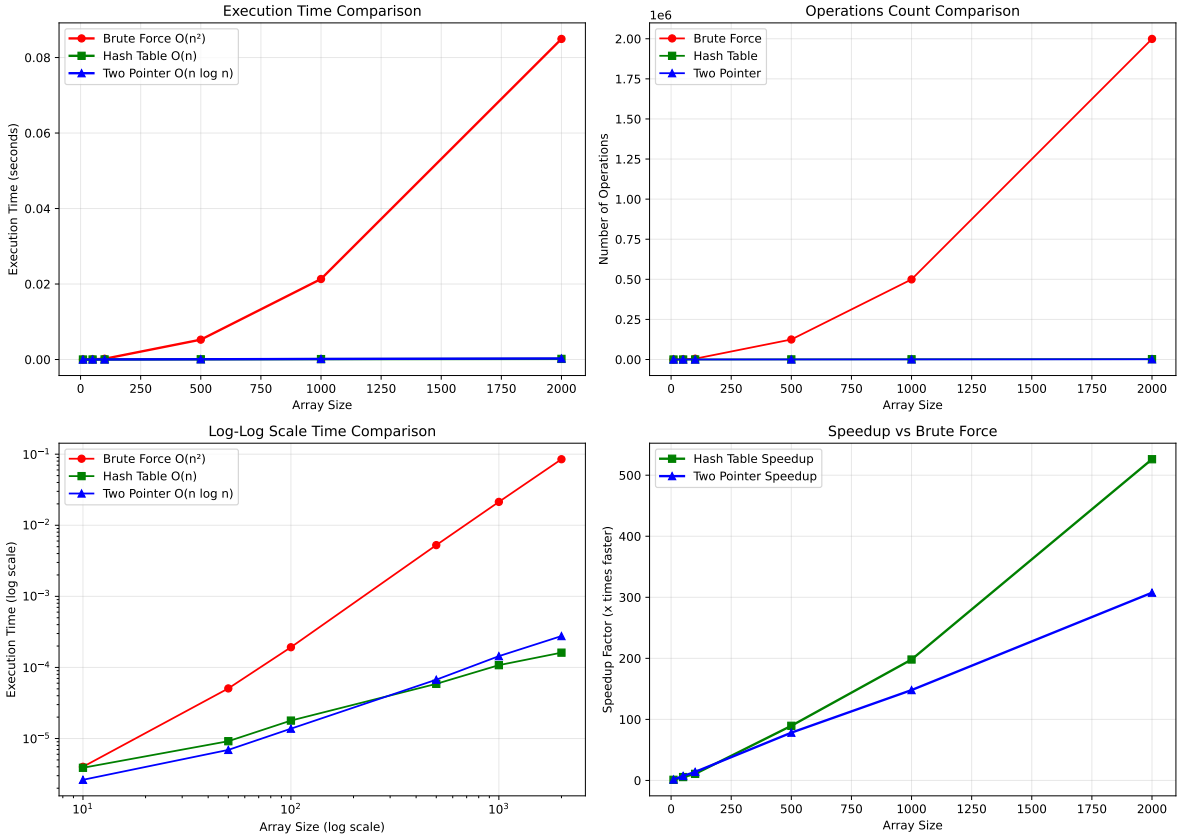
8.1 Empirical Benchmarking

```python
import time
import random
from typing import List, Dict, Tuple

def generate_test_array(size: int, zero_pairs_ratio: float = 0.3) -> List[int]:
    """Generate test array with controlled number of zero-sum pairs"""
    arr = []
    pairs_count = int(size * zero_pairs_ratio / 2)

    # Add pairs that sum to zero
    for i in range(pairs_count):
        val = random.randint(1, 100)
        arr.extend([val, -val])

    # Add random numbers (unlikely to form pairs)
    remaining = size - len(arr)
    arr.extend(random.randint(101, 200) for _ in range(remaining))

    random.shuffle(arr)
```

```
    return arr

def benchmark_solutions(sizes: List[int] = [10, 50, 100, 500, 1000, 2000]) -> Tuple[List[int]
    """Compare performance of all three approaches"""
    results = {
        'brute_force': {'times': [], 'operations': []},
        'hash_table': {'times': [], 'operations': []},
        'two_pointer': {'times': [], 'operations': []}
    }

    for size in sizes:
        arr = generate_test_array(size)

        # Benchmark brute force
        start = time.perf_counter()
        count, ops = two_sum_brute_force(arr)
        results['brute_force']['times'].append(time.perf_counter() - start)
        results['brute_force']['operations'].append(ops)

        # Benchmark hash table
        start = time.perf_counter()
        count, ops = two_sum_hash_table(arr)
        results['hash_table']['times'].append(time.perf_counter() - start)
        results['hash_table']['operations'].append(ops)

        # Benchmark two-pointer
        start = time.perf_counter()
        count, ops = two_sum_two_pointer(arr)
        results['two_pointer']['times'].append(time.perf_counter() - start)
        results['two_pointer']['operations'].append(ops)

    return sizes, results

# Run benchmarks
sizes, results = benchmark_solutions()

# Visualize results
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(14, 10))

# Execution time comparison
ax1.plot(sizes, results['brute_force']['times'], 'r-o', label='Brute Force O(n²)', linewidth=
ax1.plot(sizes, results['hash_table']['times'], 'g-s', label='Hash Table O(n)', linewidth=2)
ax1.plot(sizes, results['two_pointer']['times'], 'b-^', label='Two Pointer O(n log n)', linew
```

```
ax1.set_xlabel('Array Size')
ax1.set_ylabel('Execution Time (seconds)')
ax1.set_title('Execution Time Comparison')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Operations count comparison
ax2.plot(sizes, results['brute_force']['operations'], 'r-o', label='Brute Force')
ax2.plot(sizes, results['hash_table']['operations'], 'g-s', label='Hash Table')
ax2.plot(sizes, results['two_pointer']['operations'], 'b-^', label='Two Pointer')
ax2.set_xlabel('Array Size')
ax2.set_ylabel('Number of Operations')
ax2.set_title('Operations Count Comparison')
ax2.legend()
ax2.grid(True, alpha=0.3)

# Log scale comparison
ax3.loglog(sizes, results['brute_force']['times'], 'r-o', label='Brute Force O(n²)')
ax3.loglog(sizes, results['hash_table']['times'], 'g-s', label='Hash Table O(n)')
ax3.loglog(sizes, results['two_pointer']['times'], 'b-^', label='Two Pointer O(n log n)')
ax3.set_xlabel('Array Size (log scale)')
ax3.set_ylabel('Execution Time (log scale)')
ax3.set_title('Log-Log Scale Time Comparison')
ax3.legend()
ax3.grid(True, alpha=0.3)

# Speedup factors
speedup_hash = [bf/ht if ht > 0 else float('inf')
                for bf, ht in zip(results['brute_force']['times'],
                                  results['hash_table']['times'])]
speedup_pointer = [bf/tp if tp > 0 else float('inf')
                   for bf, tp in zip(results['brute_force']['times'],
                                     results['two_pointer']['times'])]

ax4.plot(sizes, speedup_hash, 'g-s', label='Hash Table Speedup', linewidth=2)
ax4.plot(sizes, speedup_pointer, 'b-^', label='Two Pointer Speedup', linewidth=2)
ax4.set_xlabel('Array Size')
ax4.set_ylabel('Speedup Factor (x times faster)')
ax4.set_title('Speedup vs Brute Force')
ax4.legend()
ax4.grid(True, alpha=0.3)

plt.tight_layout()
```

```python
plt.show()

# Print speedup summary
print("\nSpeedup Summary (vs Brute Force):")
print("-" * 50)
print(f"{'Size':<10} {'Hash Table':<15} {'Two Pointer':<15}")
print("-" * 50)
for i, size in enumerate(sizes):
    print(f"{size:<10} {speedup_hash[i]:<15.1f}x {speedup_pointer[i]:<15.1f}x")
```




```
Speedup Summary (vs Brute Force):
--------------------------------------------------
Size       Hash Table      Two Pointer
--------------------------------------------------
10         1.0             x 1.5           x
50         5.5             x 7.3           x
100        10.8            x 14.0          x
500        89.5            x 78.2          x
```

```
1000        198.1           x 148.0          x
2000        526.2           x 307.5          x
```

9 Expected Performance Improvements

9.1 Theoretical Analysis

Based on complexity analysis, here's what to expect:

Array Size	Brute Force $O(n^2)$	Hash Table $O(n)$	Two Pointer $O(n \log n)$	Hash Speedup	Pointer Speedup
100	4,950 ops	200 ops	~664 ops	**24.8x**	**7.5x**
1,000	499,500 ops	2,000 ops	~9,966 ops	**249.8x**	**50.1x**
10,000	49,995,000 ops	20,000 ops	~132,877 ops	**2,500x**	**376x**
100,000	~5 billion ops	200,000 ops	~1.66M ops	**25,000x**	**3,010x**

9.2 Memory Considerations

```python
from typing import List
import sys

def analyze_memory_usage(n: int) -> None:
    """Analyze memory usage for different approaches"""

    print(f"Memory Analysis for Array Size = {n:,}")
    print("=" * 50)

    # Brute force: O(1) extra space
    brute_force_extra = sys.getsizeof(0) * 3  # Just counters
    print(f"Brute Force:  {brute_force_extra:,} bytes extra")

    # Hash table: O(n) extra space
    # Worst case: all unique elements
    hash_table_extra = sys.getsizeof({}) + n * (sys.getsizeof(1) + sys.getsizeof(1))
    print(f"Hash Table:   ~{hash_table_extra:,} bytes extra")

    # Two pointer: O(n) for sorted copy (in Python)
    two_pointer_extra = n * sys.getsizeof(1)
    print(f"Two Pointer:  ~{two_pointer_extra:,} bytes extra (sorted copy)")

```

```
    print("\nSpace-Time Tradeoff:")
    print(f"  Hash Table uses {hash_table_extra // brute_force_extra:.0f}x more memory")
    print(f"  But runs {n/2:.0f}x to {n:.0f}x faster!")

analyze_memory_usage(1000)
```

```
Memory Analysis for Array Size = 1,000
==================================================
Brute Force:  84 bytes extra
Hash Table:   ~56,064 bytes extra
Two Pointer:  ~28,000 bytes extra (sorted copy)

Space-Time Tradeoff:
  Hash Table uses 667x more memory
  But runs 500x to 1000x faster!
```

10 Algorithm Selection Guide

10.1 How do I choose?

When to Use Each Approach

10.2 Use Brute Force when:

- Array is very small ($n < 50$)
- Memory is extremely limited
- Simplicity is more important than speed
- Teaching/demonstrating the problem

10.3 Use Hash Table when:

- Optimal time complexity is required
- Array is unsorted and large
- Memory usage is not a concern
- Need to handle duplicates efficiently

10.4 Use Two Pointer when:

- Array is already sorted
- Want balance between time and space
- Processing streaming/online data (can maintain sorted structure)
- Working with primitive types in low-level languages

10.5 Quick Check

1. Memory-limited, n=10,000, which approach?
   - Answer: Two-pointer (if we can afford sorting). Uses O(1) extra space vs hash table's O(n). If we can't sort in-place, then brute force despite being slow.
2. Why might two-pointer beat hash table for small arrays?
   - Answer: Less overhead. Sorting small arrays is very fast, and two-pointer has no hash function overhead or hash table creation cost.
3. 1 million transactions, which approach?
   - Answer: Hash table. $O(n) = 1$ million operations vs $O(n^2) = 1$ trillion operations for brute force. The memory for 1 million integers (~4-8MB) is acceptable on modern systems.
4. Modify for triplets?
   - Answer: This is your homework!

11 Key Takeaways

1. **Algorithm Choice Matters:** $O(n^2)$ vs $O(n)$ makes a massive difference at scale
2. **Hash Tables are Powerful:** Trading space for time often yields huge speedups
3. **Understand Your Data:** Sorted data enables different optimizations
4. **Handle Edge Cases:** Zeros and duplicates need special attention
5. **Profile Before Optimizing:** Measure actual performance, not just complexity

12 Further Reading

- Hash Table Deep Dive
- Two Pointer Technique Patterns
- Counting Problems in Competitive Programming
- Python's Dictionary Implementation