COMP101 — Week 7
Problem Solving with Algorithms

from ideas to code

UM6P — SASE

November 28, 2025

Learning goals

By the end of this week, you should be able to:

- **Explain** what an algorithm is and describe it in plain language and pseudocode.
- **Apply** basic algorithmic patterns (scan, accumulate, filter, map, frequency count, sets).
- **Translate** these patterns into clear Python functions using lists, sets, and dictionaries.
- **Use** a simple workflow to go from problem statement to working code.
- **Reason informally** about why some algorithms are faster than others (binary search vs naive search).

Warm-up: Guess the number

- I choose a secret integer between 1 and 100.
- You propose guesses; I answer `too high`, `too low`, or `correct`.
- Goal: find the number using as few guesses as possible.

Warm-up: Guess the number

- I choose a secret integer between 1 and 100.
- You propose guesses; I answer `too high`, `too low`, or `correct`.
- Goal: find the number using as few guesses as possible.

Algorithm

Question for you: What strategy guarantees that we always find the number in at most a small number of guesses?

Algorithm

Strategy (informal):

1. Keep track of the current range `[low, high]` where the secret number can be.
2. Always guess the middle: `mid = (low + high) // 2`.
3. If the answer is `too high`, move the high bound: `high = mid - 1`.
4. If the answer is `too low`, move the low bound: `low = mid + 1`.
5. Repeat until we hit `correct`.

This is an **algorithm**: a clear, step-by-step method that always works for this task.

What is an algorithm?

Definition

An **algorithm** is a finite, ordered sequence of unambiguous steps that transforms **input** into **output** and eventually **terminates**.

Examples:

- A recipe for a cake.
- Instructions for logging into the university Wi-Fi.
- Sorting a hand of cards by repeatedly inserting each card in its place.

Our Python toolbox so far

We already know how to:

- Use **variables**, **expressions**, and **conditions** (`if`, `elif`, `else`).
- Write **loops** (`for`, `while`) to repeat work.
- Define **functions** to organize and reuse code.
- Work with **lists**, **sets**, and **dictionaries**.

Today is about combining these tools into reusable **problem-solving patterns**.

Algorithm

General workflow

1. **Understand** the problem: inputs, outputs, and constraints.
2. **Explore** small examples and edge cases.
3. **Design** an algorithm in words or pseudocode.
4. **Translate** the algorithm into Python code.
5. **Test** the solution with normal, edge, and "evil" cases.

We will follow this workflow for each pattern.

Pattern 1: scanning / linear search

Definition

Task: given a list and a target value, decide whether the target is in the list, and where.

```
def find_first(a, target):
    """Return index of first target in a, or -1 if not found."""
    for i in range(len(a)):
        if a[i] == target:
            return i
    return -1
```

- Visits each element once, in order.
- Works on any sequence type, no sorting needed.

Pattern 2: accumulation

Definition

Many problems are: **go through all items and combine them** into a single result (sum, count, min, max, average, ...).

```
def average(nums):
    total = 0
    for x in nums:
        total += x
    return total / len(nums)
```

Pattern 2: accumulation

Definition

Many problems are: **go through all items and combine them** into a single result (sum, count, min, max, average, ...).

```
def average(nums):
    total = 0
    for x in nums:
        total += x
    return total / len(nums)
```

```
def count_positive(nums):
    count = 0
    for x in nums:
        if x > 0:
            count += 1
    return count
```

Pattern 3: filtering and mapping

Filtering: keep only elements that satisfy a condition.

```
def only_even(nums):
    result = []
    for x in nums:
        if x % 2 == 0:
            result.append(x)
    return result
```

Pattern 3: filtering and mapping

Filtering: keep only elements that satisfy a condition.

```
def only_even(nums):
    result = []
    for x in nums:
        if x % 2 == 0:
            result.append(x)
    return result
```

Mapping: transform each element.

```
def square_all(nums):
    result = []
    for x in nums:
        result.append(x * x)
    return result
```

Pattern 4: frequency counting with dictionaries

Definition

We often need to know **how many times** each item appears (characters, words, grades, IDs).

```
def word_counts(text):
    counts = {}
    for word in text.split():
        word = word.lower()
        if word not in counts:
            counts[word] = 0
        counts[word] += 1
    return counts
```

Pattern 4: frequency counting with dictionaries

Definition

We often need to know **how many times** each item appears (characters, words, grades, IDs).

```
def word_counts(text):
    counts = {}
    for word in text.split():
        word = word.lower()
        if word not in counts:
            counts[word] = 0
        counts[word] += 1
    return counts
```

- Keys: the distinct words.
- Values: how many times each word appears.

Pattern 5: sets and uniqueness

Definition

A **set** stores unique elements and supports very fast membership tests.

```
def num_distinct(nums):
    unique = set()
    for x in nums:
        unique.add(x)
    return len(unique)
```

Definition

A **set** stores unique elements and supports very fast membership tests.

```
def num_distinct(nums):
    unique = set()
    for x in nums:
        unique.add(x)
    return len(unique)
```

- Useful when you only care whether something appears, not how many times.
- Example: how many distinct students attended at least one lab?

Mini case study: simple text analytics

Definition

Given a piece of text, compute:

- total number of words,
- number of distinct words,
- the most frequent word.

```
def text_stats(text):
    counts = word_counts(text)      # reuse previous function
    total_words = sum(counts.values())
    distinct = len(counts)
    most_word = max(counts, key=counts.get)
    return total_words, distinct, most_word
```

We are now composing small algorithms into a more complex one.

Binary search vs naive guessing (intuition)

- Naive: guess randomly or move one by one; in the worst case you may check many numbers.
- Binary search: each guess halves the remaining interval.

Binary search vs naive guessing (intuition)

- Naive: guess randomly or move one by one; in the worst case you may check many numbers.
- Binary search: each guess halves the remaining interval.

Theorem / Fact

If you always guess the middle in the range 1–100, you need at most about 7 guesses. (After each guess, the remaining interval size is roughly divided by 2.)

We will formalize searching and sorting better in Week 8, but you already know the core idea.

Recap: algorithm patterns cheat sheet

Today we:

- Defined what an **algorithm** is.
- Introduced a general **problem-solving workflow**.
- Practiced core patterns:
  - scanning / linear search,
  - accumulation (sum, count, min, max),
  - filtering and mapping,
  - frequency counting with dictionaries,
  - sets for uniqueness.
- Saw mini case studies (phonebook, text analytics).
- Used the **guess-the-number** game to motivate binary search.