COMP101 — Week 7
Problem Solving with Algorithms

Searching and Sorting

UM6P — SASE

December 5, 2025

Searching and Sorting

Searching and Sorting

- Today: how to **find things fast** and how to **prepare data** for fast searching.
- We start from a simple game: **guess the number**.
- This will lead us to:
  - **Binary search** on ordered data.
  - **Sorting** as a way to enable fast search.
  - Three basic sorting algorithms: **bubble**, **selection**, **insertion**.
  - How to use Python's `sorted()` and `list.sort()`.

Learning Objectives

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

- Explain the idea of **binary search** and why it is efficient.
- Explain why **sorting** helps with searching.
- Describe, in your own words, the ideas of:
  - **Bubble sort**
  - **Selection sort**
  - **Insertion sort**
- Give a rough intuition of algorithms with time complexity:
  - $O(n)$, $O(n^2)$, $O(n \log n)$.
- Use **`sorted()`** and **`list.sort()`** correctly in Python.

Warm-up Game: Guess the Number

Game: Guess the Number

- I am thinking of a number between **1 and 100**.
- Your goal: **find my number using as few questions as possible**.
- Two phases:
  - Phase 1: **naive strategy** (random guesses).
  - Phase 2: **smart strategy** with restricted questions.

Naive Strategy

- Typical naive approach:
  - "Is it 37?"
  - "Is it 52?"
  - "Is it 11?"
- In the worst case you might have to try **almost every number**.
- This is similar to **linear search**:
  - Try each possibility one by one.
  - Worst case: $O(n)$ questions.

Restricted Questions

- New rule: you can only ask **yes/no questions** of the form:
  - "Is your number $> k$?"
  - "Is your number $< k$?"
  - "Is your number $= k$?"
- Question: **What strategy should we use now?**
- Idea: always choose the **middle** of the remaining interval.
  - Start with 1–100.
  - First question: "Is your number > 50?"
  - Then keep only the half that is still possible.

Halving the Interval

- Each question **cuts the search interval roughly in half**:
  - 1–100
  - 1–50 or 51–100
  - 1–25 or 26–50 or 51–75 or 76–100
- Number of questions grows like $\log_2(n)$, not like n.
- This is the core idea of **binary search**.
- Key requirement: the possible values are **ordered**.

Binary Search

Binary Search on a Number Line

Definition

Binary search repeatedly **splits** the search range in two, then **keeps only the half** that can still contain the target.

- We have a range of numbers: **1 to 100**.
- We want to know if a target number is in that range.
- Binary search strategy:
  - Look at the **middle**.
  - Decide if the target is on the left half or the right half.
  - **Discard** the half where the target cannot be.
  - Repeat on the remaining half.
- This is a classic **divide and conquer** idea.

Binary Search on a Sorted List

- Example list: $[2, 5, 7, 10, 14, 18, 21]$ (**sorted**).
- Target: **14**.
- Steps:
  - Check middle: **10**.
  - 14 is larger than 10, so we discard the left half including 10.
  - New list part: $[14, 18, 21]$.
  - Check middle: **18**.
  - 14 is smaller than 18, so we keep $[14]$.
  - Found **14**.
- Very few comparisons compared to checking every element.

Requirements for Binary Search

- The list (or range) **must be sorted**.
- We must be able to **compare** elements to the target.
- We must be able to access the **middle position** quickly (arrays, lists).
- If the list is **not sorted**, **binary search does not work**.

Unsorted Lists and Linear Search

What if the List is Not Sorted?

- Consider the list: $[17, 42, 3, 99, 8, 11]$ (**unsorted**).
- We want to find **42**.
- If we look at the "middle" element (e.g. 99), what can we conclude?
  - We cannot say "42 is only on the left" or "only on the right".
  - The left side is not guaranteed to be smaller than the right side.
- **Binary search fails** on unsorted lists.

Linear Search

Definition

Linear search checks elements **one by one** from start to end until it finds the target or reaches the end.

- Simple alternative for unsorted data.
- Worst case:
  - You inspect **every element**.
  - Time complexity: $O(n)$.
- Good enough for:
  - **Small lists**.
  - **Rare searches**.
- But if we search **many times**, we can do better by preparing the data.

Why Sorting Matters

Dictionary Analogy

- How do you look up a word in a **dictionary**?
  - Words are **sorted alphabetically**.
  - You do not read page 1, then 2, then 3.
  - You "jump around" based on alphabetical order.
- The dictionary has been **sorted once**.
- Sorting allows you to perform something like **binary search with your eyes**.

Mess vs Order

- Imagine:
  - A **messy stack** of unsorted papers.
  - A **neatly sorted** filing system with labeled folders.
- In both cases, the information is the same.
- Difference: how **fast** you can find what you need.
- Sorting is like creating that **neat filing system**.

Sorting as Preprocessing

- Sorting has a **one-time cost**.
- After that, searches can be much faster:
  - Unsorted data + linear search: $O(n)$ per search.
  - Sorted data + binary search: $O(\log n)$ per search.
- If you search **many times**, sorting can be a very good trade-off.

Bubble Sort

Bubble Sort: Idea

Algorithm

Bubble sort repeatedly **compares neighboring elements** and **swaps** them if they are in the wrong order. Large elements "**bubble up**" to the end.

- Start with a list, for example: $[5, 1, 4, 2]$.
- Compare each pair of **neighbors**:
  - Compare 5 and 1: out of order, so **swap**.
  - Compare 5 and 4: out of order, so **swap**.
  - Compare 5 and 2: out of order, so **swap**.
- After one full pass, the **largest** element has "bubbled up" to the end.
- Repeat passes until the list is sorted.

Bubble Sort: High-level Pseudo-code

Algorithm

- Repeat the following passes:
  - Go through the list from the start to the second-to-last element.
  - For each index j, compare element j with $j + 1$.
  - If they are in the wrong order, **swap** them.
- Stop when a pass makes **no swaps**.

Bubble Sort: Pros and Cons

- Pros:
  - **Very simple**.
  - Easy to implement and visualize.
- Cons:
  - **Very slow** for large lists.
  - Worst-case time complexity: $O(n^2)$.
- Mainly used for **teaching**, not in real-world large-scale code.

Selection Sort

Selection Sort: Idea

Algorithm

Selection sort repeatedly **selects the smallest element** from the unsorted part and **places it** into its correct position.

- Think in terms of positions in the list.
- Step 1:
  - Find the **smallest** element in the list.
  - Put it in **position 0** (swap if needed).
- Step 2:
  - Among the remaining elements, find the smallest.
  - Put it in **position 1**.
- Repeat for positions 2, 3, ... until the list is sorted.

Selection Sort: High-level Pseudo-code

Algorithm

- For index i from 0 to $n-1$:
  - Assume the minimum element is at position i.
  - Search from $i+1$ to $n-1$ for a **smaller** element.
  - If you find one, remember its index as the **new minimum**.
  - At the end of the scan, **swap** element at i with the smallest found.

Selection Sort: Pros and Cons

- Pros:
  - Simple idea.
  - Performs relatively **few swaps**.
- Cons:
  - Still scans the rest of the list for each position.
  - Time complexity: $O(n^2)$ comparisons.
- Again, mainly **educational** for understanding sorting concepts.

Insertion Sort

Insertion Sort: Card Analogy

Algorithm

Insertion sort keeps a **sorted prefix** and **inserts** each new element into its correct position inside that prefix.

- Imagine sorting playing cards in your hand.
- You start with one card: it is already "sorted".
- For each new card:
  - You look from right to left in your hand.
  - You insert the new card in the **correct place**.
- This is exactly the idea of **insertion sort**.

Insertion Sort: Algorithm Idea

Algorithm

- Consider the first element as a **sorted sublist**.
- For each index i from 1 to $n - 1$:
  - Save the element at i as **key**.
  - Compare key with elements to its left.
  - **Shift** larger elements one position to the right.
  - **Insert** key in the correct position.

Insertion Sort: Pros and Cons

- Pros:
  - **Very intuitive**.
  - Efficient for **small lists**.
  - Efficient when the list is **almost sorted** (best case $O(n)$).
- Cons:
  - Worst-case time complexity: $O(n^2)$.
  - Still not good for very large, randomly ordered lists.

Complexity Intuition

Rough Operation Counts

Theorem / Fact

Bubble sort, selection sort and insertion sort all perform **on the order of n^2** basic operations in the **worst case**.

- Simple table:
  - $n = 10 \Rightarrow n^2 = 100$ operations (roughly).
  - $n = 100 \Rightarrow n^2 = 10\,000$ operations.
  - $n = 1000 \Rightarrow n^2 = 1\,000\,000$ operations.
- When n is multiplied by **10**, n^2 is multiplied by **100**.

Comparing Growth Rates

- **Linear**: $O(n)$
  - Example: **linear search**.
- **Quadratic**: $O(n^2)$
  - Example: **bubble**, **selection**, **insertion** sort (worst case).
- **"Better" sorts**: $O(n \log n)$
  - Examples (names only): **merge sort**, **quicksort**.
  - Used in **real libraries** and in Python's implementation.
- For large n, $n \log n$ grows much slower than n^2.

Sorting in Python

The sorted() Function

- **`sorted(iterable)`**:
  - Takes any iterable (list, tuple, string, set, range, ...).
  - Returns a **new list** that is sorted.

The `list.sort()` Method

- Lists have a **`sort()`** method:
- Important:
  - `list.sort()` **modifies the list in-place**.
  - It **returns `None`**.
  - Common bug: **`x = x.sort()`** results in **`x` being `None`**.

Sorting with key and reverse

- Both `sorted()` and `list.sort()` accept:
  - `reverse=True` for descending order.
  - `key=function` to choose what to sort by.

Quick Exercises

Exercise: Predict Bubble Sort

- Given the list $[5, 1, 4, 2]$, what is the list after:
  - **One full pass** of bubble sort?
- Discuss:
  - Which element is guaranteed to be in the **correct position**?
- Purpose:
  - Check your understanding of how **bubble sort** moves elements.

Exercise: Recognize the Algorithm

- I show you a short piece of pseudo-code that:
  - Scans the list to find the **smallest element**.
  - Swaps it into **position 0**.
  - Repeats for positions 1, 2, ...
- Question:
  - Is this **bubble sort**, **selection sort**, or **insertion sort**?
- Similar exercises can be done for **insertion sort**.

Concept Check

- When is **linear search** acceptable?
  - Small lists, rare searches, or one-off tasks.
- Why can we **not** use **binary search** on an **unsorted** list?
  - The middle element does not tell us which half to discard.
- What is the main difference between:
  - `sorted(my_list)`
  - `my_list.sort()`

Summary and Conclusion

Summary

- We started with a guessing game and discovered **binary search**.
- We saw that binary search **requires sorted data**.
- We introduced three basic sorting algorithms:
  - **Bubble sort**
  - **Selection sort**
  - **Insertion sort**
- All three have **worst-case** time complexity $O(n^2)$.
- In practice, we rely on Python's built-in **`sorted()`** and **`list.sort()`**.