CS1301x Revision Note

Computing in Python IV:

Objects & Algorithms

(unoffical personal note)

kayikc @

December 14, 2024

Contents

1 Algorithms

Computers excel at quickly performing complex math. Machine learning, for instance, combines statistics with powerful hardware to handle massive calculations. This exemplifies the "algorithmic" side of computer science. An algorithm is simply a defined series of steps transforming input into output.

Examples in Daily Life:

- Data compression
- Random number generation
- Search algorithms

2 Complexity and Big O Notation

The power of automated algorithms lies in chaining steps to solve otherwise impractical problems. Let's consider searching names in a school roster:

- **Sorted:** Finding "David Joyner" takes fewer than 10 checks using binary search.
- **Unsorted:** Might require up to 500 checks, one by one.

Finding duplicates is even trickier:

- **Sorted:** Around 500 checks (comparing neighbors).
- **Unsorted:** Could take 125,000 checks (every name against all others).

Inefficient algorithms become dramatically slower as data grows. Doubling the students quadruples the work!

Big O notation ($O(\cdot)$) measures how an algorithm's workload increases with data size (n). It focuses on the dominant growth factor, ignoring smaller details like constant multipliers. For instance, $n^2/2$ simplifies to $O(n^2)$. We care about the scaling trend, not minor optimizations.

Common Complexities (from fastest to slowest):

- $O(1)$ (Constant): Same time, regardless of data size (e.g., checking the first list item).
- $O(\log n)$ (Logarithmic): Gets relatively faster with larger input (e.g., binary search in a sorted list).
- $O(n)$ (Linear): Time grows directly with input (e.g., checking each item once).
- $O(n^2)$ (Quadratic): Time grows with the square of the input (e.g., duplicate checks in an unsorted list).
- $O(2^n)$ (Exponential): Time doubles with each additional input (e.g., password brute-forcing).

Lower complexity means a faster algorithm, especially for large datasets.

3 Recursion

The way I think about Recursion:

1. Find the base case (simplest form): Identify the smallest, simplest input(s) for which you can directly compute the answer without recursion.

2. Find the general equation (recursive step): Determine how to express the solution for larger inputs in terms of the solution(s) for smaller, self-similar subproblems.

3.1 Example: Finding a Route

Imagine a function `find_route(start, end)` that finds a path from a starting point to an end point. Recursively, this might be represented as follows:

Listing 1: Conceptual Example: Recursive Route Finding

```
find_route(front_door, office_door)

# Breaks down into:
find_route(front_door, stairwell_entrance)
find_route(stairwell_entrance, stairwell_exit)
find_route(stairwell_exit, office_door)

# ...and further:  % Improved comment style
find_route(stairwell_exit, hallway)
find_route(hallway, office_door)
```

The function calls itself with progressively simpler subproblems, eventually reaching a base case (e.g., the start and end points are the same location), where a direct route exists.

3.2 Simple Example: Factorial

The factorial of a number n (written $n!$) is the product of all positive integers up to n: $n! = n \times (n-1) \times \cdots \times 1$. The recursive definition is $n! = n \times (n-1)!$ with a base case of $1! = 1$.

Listing 2: Recursive Factorial

```
def factorial(n):
    # If n is greater than 1, multiply n by factorial of (n-1)
    if n > 1:
        return n * factorial(n - 1)
    else:
```

3.3 Intermediate Example: Fibonacci Series

The Fibonacci sequence starts with 1, 1 and continues with each number being the sum of the previous two: 1, 1, 2, 3, 5, 8, and so on. Recursively:

- **Function** `fibonacci(n)`:
- **If** $n > 2$: **return** `fibonacci(n-1) + fibonacci(n-2)`
- **If** $n \leq 2$: **return** 1

This simple recursive implementation is inefficient as it recalculates values multiple times (e.g., 'fibonacci(4)' calculates 'fibonacci(3)' twice). A more efficient approach uses memoization (storing calculated values in a dictionary) to avoid redundant calculations.

3.4 Advanced Example: Directory Exploration

Listing all files in nested directories can be done recursively:

- **Function** `listFiles(directory)`:
- List each file in `directory`.
- For each subfolder in `directory`, call `listFiles(subfolder)`.

This mimics exploring a building: list items in the current room, then open each door and repeat for the new rooms.

Head vs. Tail Recursion: Head recursion processes the current level before going deeper (root → subfolder → deeper subfolder). Tail recursion processes the deepest levels first, then the current level (deepest → subfolder → root).

4 Sorting Algorithms

Sorting algorithms arrange items in a list into a specific order, such as ascending order. They take an unsorted list as input and produce a sorted version as output. These algorithms can sort various data types (numbers, strings, etc.) as long as there's a way to compare elements. Many sorting algorithms exist, each with varying efficiency. We'll use a 10-number list as a running example, prioritizing ease of implementation over computational complexity.

Figure 1: The list of 10 unsorted numbers

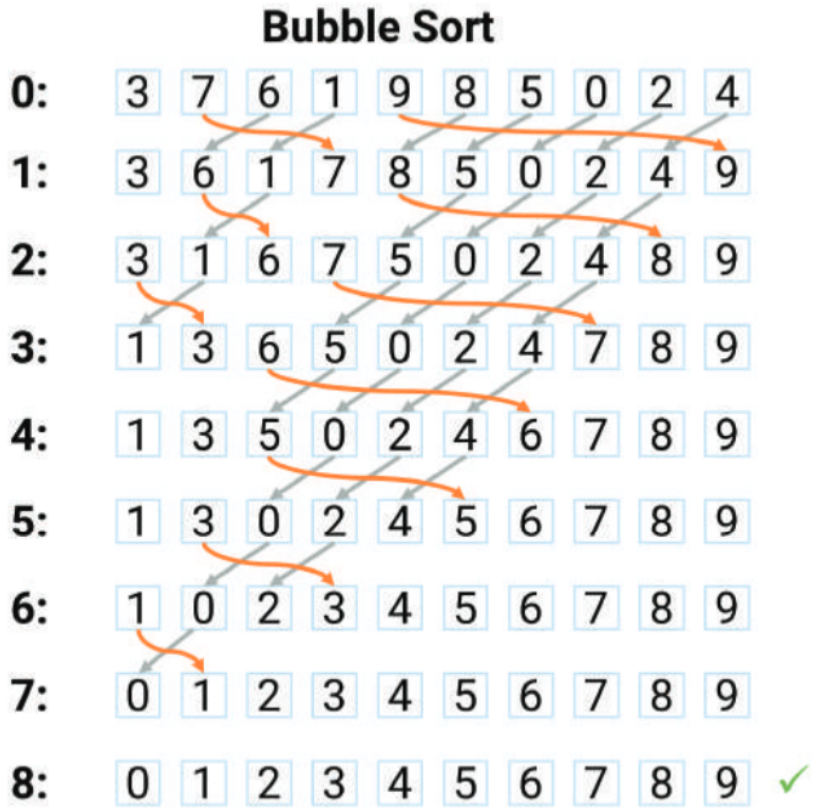

Figure 2: Example of Bubble Sort. Orange arrows indicate comparisons and swaps.

4.1 Bubble Sort

‘’

Bubble sort is a straightforward yet inefficient algorithm. It repeatedly compares adjacent pairs of numbers, swapping them if they’re in the wrong order. This process continues until a pass through the list requires no swaps, indicating the list is sorted.

Let’s walk through the example in Figure 2. The initial list is 3 7 6 1 9 8 5 0 2 4.

1. **Pass 1:** The algorithm compares 3 and 7 (no swap needed), then 7 and 6 (swap!), 6 and 1 (swap!), 1 and 9 (no swap), and so on. The largest number, 9, "bubbles" to the end.

2. **Pass 2:** The process repeats. The second largest number, 8, moves closer to its final position.

3. **Subsequent Passes:** This continues. Each pass places the next largest unsorted number in its correct position. Notice how smaller numbers gradually "float" towards the beginning (e.g., 0 moves one step forward with each pass).

The orange arrows in Figure 2 show the comparisons and swaps. The algorithm continues until a pass requires no swaps (the last two rows are identical, showing the sorted list).

Efficiency: Bubble sort has a time complexity of $O(n^2)$. This means it can be very slow for large lists. In the worst-case scenario (a reverse-sorted list), it performs roughly n^2 comparisons.

4.2 Other Sorting Algorithms (Overview)

To do

5 Search Algorithms

To do

5.1 Linear Search

To do

5.2 Binary Search

To do

5.3 Linear vs. Merge Sort + Binary Search

To do