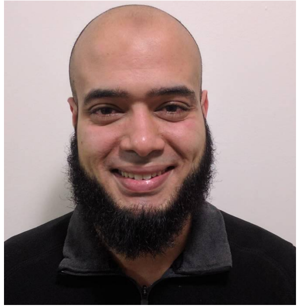
Algorithms

Insertion Sort 2

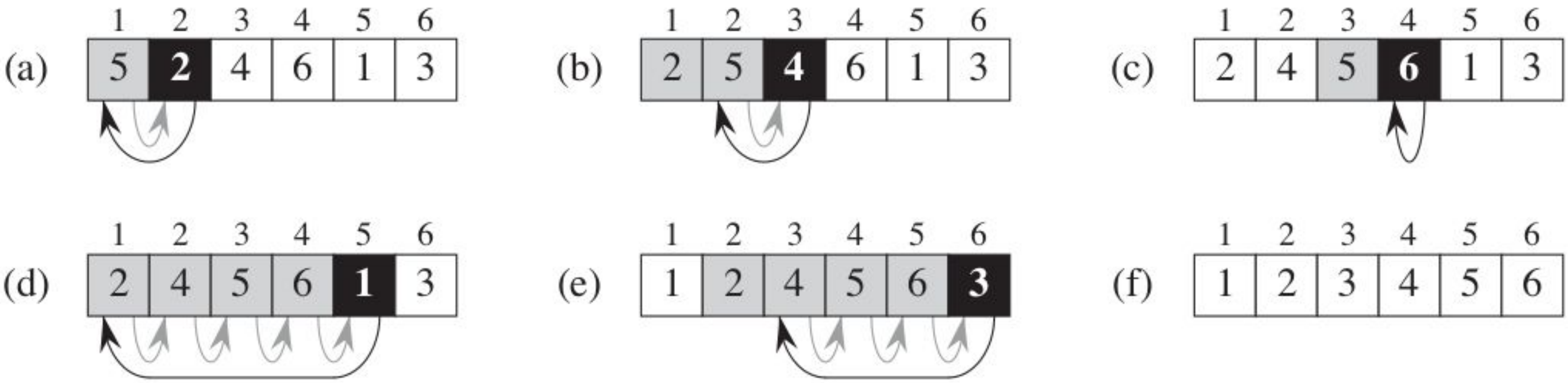
Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



How to code?

- How to insert the number in its right place?
  - Find the right place
  - Shift the right-sub array one step right then insert the current element
  - We can do these 2 steps together!



Code: O(n^2) time and O(1) space

```
void insertion_sort(vector<int> &nums) {
    // For each number: add it in the previous sorted subarray
    for (int i = 1; i < (int) nums.size(); i++) {
        int key = nums[i];
        int j = i - 1;
        // Shift and add in the right location
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j];  // shift right
            j--;
        }
        nums[j + 1] = key;  // The first right element
    }
}
```

Testing

- It is very critical to test your code
- Think in a systematic way about possible cases
- Here are general thoughts that may help in different cases for arrays
- Length: 1, 2, 3, and the max N
    - Boundaries can be tested with the smallest N and the largest N
- Values:
    - Odd and even
    - Duplicate arrays or unique values
    - Sorted, Almost sorted, not ordered (partially, completely)
- In this problem what matters:
    - Length: 1 and N. Test a random array. Test some arrays with duplicate values

Some observations

- Assume the array is (almost) sorted: e.g. 1, 2, 3, 4, 5, 6, 7
  - What is the time complexity?
- Clearly, the 2nd nested loop will never work. So we are actually O(n)
  - This is called the **Best-case** performance (behavior under optimal conditions)
- What if the array is already ordered from large to small? E.g. 7, 6, 5, 4, 3, 2, 1
- Clearly, the 2nd nested loop is applied to the last index. This is O(n^2).
  - This is called the **Worst-case** performance
- The average case is O(n^2)
  - The second loop is applied (partially or fully) most of the time

Worst vs Best analysis

- Many algorithms with bad worst-case performance have good average-case performance
  - If this is the case, most of the time your code will be pretty fast **except for a few cases**!
- Some data structures like **hash tables** have *very poor* worst-case behaviors, but a **well written** hash table of **sufficient size** will never give the worst case
  - That is: A good implementation + proper usage.
- Take-home message: Don’t be systematic when computing/using such types of analysis. In practice, we *might* need to think deeper about what the **typical inputs** are and the **effect** of that on the problem of interest

Correctness

- We must prove the correctness of our approach too
- Many books go very formal for a few pages explaining the proof.
  - Reading lengthy proofs can be exhausting to many people unless you have a good mathematical background
  - Understanding proofs is still a very added value. It teaches you to make sure your logic covers the possible scenarios
  - You need to read formal proofs to be able to write one, if you are interested
- One such book is Introduction to Algorithms (CLRS) by Cormen et al.
  - I learned from it. It focuses on the theory and is very mathematical but a **great book**
- There are other books which focus more on the practical side, such as Algorithms Design Manual *by Steven Skiena*

Correctness

- I want to tell you a few things here
- I suggest a flow that will make your progress **faster** and much more **productive**
  - First, focus a lot on the **intuition** behind the approach
    - Strong algorithmist can find solutions for very hard problems *in a few minutes*
  - Understand the **code**. Think in test cases. Build **informal thoughts** about correctness
  - **Solve** several exercises about the algorithm
  - Optional: Check out the **proof**.
    - But, at the very least, please find and read the proofs for a few of them

Correctness

- Insertion sort: Informal proof
  - I hope you can trivially see why it is a correct algorithm
  - At n = 1, the initial sub-array of A[0] is sorted
  - Then from n = 2, covering **all** the elements. We search linearly to find the correct location
    - Then we shift its right subarray, where all the shifted values are > current key
    - This means: after the ith step, the extended subarray is sorted

Insertion Algorithm Properties

- Sorting algorithms have interesting properties to understand
- Adaptive, i.e., efficient for data sets that are already substantially sorted
  - The time complexity is O(kn) when each element in the input is no more than k places away from its sorted position. If the whole list is already sorted, then k = 1
- Stable; i.e., does not change the relative order of elements with equal keys
  - Assume input [1, 2, 5A, 5B, 3, 5C]. [A,B,C just tags]
  - When we sort it: [1, 2, 3, 5A, 5B, 5C]. Equal keys have the SAME old order
- In-place; i.e., only requires a constant amount O(1) of additional memory space.
  - As you see, we were updating the given vector itself
- Online; i.e., can sort a list as it receives it
  - Imagine an **online service** that keeps receiving numbers and sort all what we have **so far**

Comparison based algorithms

- As you notice, this algorithm, and many others, **compares** numbers together to find out the right output
  - We will meet other efficient comparison-based algorithms
    - E.g. Merge sort is only O(n logn)
- We will also learn other types of algorithms that are **not** based on comparisons (e.g. Count sort)
- An interesting fact to know: any **comparison-based** sorting algorithm must make at least ($n \log_2 n$) comparisons *on average* to sort the input array
  - So never try to find something better :)
  - That is: Sometimes we compute the **lower-bound** complexity for an algorithm
- *Sorting is the most thoroughly studied problem in computer science.*

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”