1.1 Course information

- My contact info: Vijay Chaudhary, vijay dot chaudhary dot phd at gmail dot com, office hours Monday & Wednesday 3-4 pm at **TBA**.
- TA Office hours: TBA

Here are the grading policies:

- Quizzes (5% of grade):
  - **∼10 quizzes**
  - There will be a quiz at a random time during lecture once a week.
  - It will be based on previous lectures.
- Homework (45% of grade):
  - **6 homeworks** (+ Homework 0)
  - Students will have a week to complete the homework assignments once they are released. Homeworks will be submitted on Canvas. Printed or handwritten copy of homework submissions will also be accepted.
  - **No late submission will be accepted.** One of the homeworks with the lowest grade will be dropped. You should still attempt all of the assignments.
  - You may discuss approaches to solving a problem with others, however, your submission must be written on your own words. **Plagiarized submissions will be zero-ed out.**
  - Homework 0 has already been posted on the course website. You will receive extra credit points for this work. These are fairly easy problems for someone who has completed the prerequisites for this course.
- Midterm Exams (30% of grade):
  - There will be 2 midterm exams evenly spaced out during the semester. Exam 1 will be on **Wednesday, Sept. 20**, Exam 2 will be on **Monday, Oct. 23**.
- Final Exam (20% of grade): Final Exam will be during the finals week. The schedule for the final exam will be scheduled by the department.

1.2 Why are you here?

It's fair to say that you are mostly here because it is a required course, but there must be a reason, you are required take it.

1. **Algorithms is fundamental to computer science.** Although computer science is usually introduced with writing computer programs in various programming languages such as Python, C++, etc., one may immediately realize algorithms an important aspect of writing computer programs. Implementing a computer program will almost always require an algorithm or several algorithms. All advance computer science courses such as Operating Systems, Networking, Machine Learning, Cyber Security, and Mobile App Development heavily use the elements of algorithms we will discuss in this course.

   Algorithms is useful across non-CS disciplines (economics, biology, psychology), primarily because several non-CS questions are addressed computationally. For instance, writing simulations are central to studies done in psychology to model human behaviors.

2. **Algorithms is useful.** Sorting and searching are something we are always involved in our daily lives. Organizing tasks requires some implicit sorting. Imagine searching for a book in an unsorted stack in a library or searching for a food item in a large grocery store without organization! I am merely pointing out the presence of algorithms used in our daily lives.

   This course is mostly about algorithms used in computations. As a computer science professional, algorithms will be useful in your entire professional life. While you may not be required to write time/space complexity analyses all the time, you may require to have gauge the efficiency of an algorithm, and trade-offs between algorithms to accomplish the same task.

   Most programming interviews ask algorithm questions. A programming interview typically requires you to come up with an (efficient) algorithm for a computational problem, implement the algorithm using a programming language, and then state the time/space complexity of the algorithm to tell how efficient the algorithm is.

3. **Algorithms is interesting and fun!** As we progress through the course, we will observe how some algorithms are efficient than the others tackling the same problem. For instance, ***insertion sort*** is something we intuitively use in our daily lives, which roughly takes $c_1 n^2$ amount of time for n items. Meanwhile, ***merge sort*** takes roughly $c_2 n \lg n$ amount of time. Say, $c_1 < c_2$, at face value, ***insertion sort*** has a better performance compared to ***merge sort*** for smaller value of $n \leq n_0$, however, ***merge sort*** will **always** perform better than ***insertion sort*** for a larger value of $n > n_0$. It's a mathematical certainty, (which is kind of a big statement to make). We will do comparison with calculations to show their performance differences today.

1.3 Algorithms as a technology

Say, we had infinitely fast computers and computer memory were free. Would we still need algorithms? The answer is yes. Even in such utopic scenario, **algorithms would be required to determine if a computer program was going to terminate and give**

us the correct answer. You can see that you'd be still required to take this course even if the world had infinitely fast computers.

We all know the computers we have are not infinitely fast, and there is a limited computer memory at disposable. In other words, we have bounded computing resources, and we want to make the best out of them.

1.3.1 Efficiency

Let's compare the efficiency of two sorting algorithms with different runtime complexities on a hardware.

Comparing Insertion Sort and Merge Sort		
	Computer A	Computer B
Algorithm	Insertion Sort	Merge Sort
Computing Power	10 billion instructions per second	10 million instructions per second
Computer Program Quality	highly efficient	average
Runtime to sort n numbers	$2n^2$	$50n \lg n$
size of array; n	10 million $= 10^7$	10 million $= 10^7$
# of instructions to sort n numbers	$2 * (10^7)^2$	$50 * (10^7) * \lg(10^7)$
Sorting time in seconds	$\frac{2*(10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$	$\frac{50*(10^7)*\lg(10^7) \text{ instructions}}{10^7 \text{ instructions/second}}$
	$20,000$ seconds	≈ 1163 seconds
	5.56 hours	19.38 minutes

What if we have 100 million numbers to sort? Can you use the calculations above to find the runtime difference between insertion sort and merge sort for the computer A and computer B?

1.4 Class Exercise

What is the smallest value of n, say n_0, such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Note: Later we will see why this threshold value, n_0, is essential to find the asymptotic bounds.

1.5 Insertion Sort

Let's look at a sorting algorithm, and do a runtime analysis on it. We will also use this algorithm to look at **loop invariant** and **correctness of an algorithm** with ***insertion sort***.

Sorting is widely used in applications we use in our daily lives. For instance, sorting items by low to high prices, or high to low prices, sorting contacts list, etc. Among several sorting algorithms, we will work with insertion sort for it's something we intuitively use to sort items in small numbers.

A good example to demonstrate insertion sort is the way we sort a set of cards dealt in a card game. Once a set of n cards is dealt to us,

1. we pick a card, which is sorted trivially.
2. as we continue picking the cards from the dealt pile on the table, we find the insertion index among the sorted set of cards on our hand to put the picked card.
3. once we there is no more card left on the table to pick, we are left with a set of sorted cards on our hand.

We will identify loop invariant in this procedure in a bit.

The figure below demonstrates insertion sort in an array step by step. We start with an unsorted array. Then, each step is pivoted to a **key**, with a sub-array left to the key is always sorted, while right to the key is unsorted. The first number of the array will be sorted, therefore, we start with the key indexed at 2. For each key, we find a correct insertion index in the left sorted array, and insert the key to have a sorted left sub-array after the insertion. We keep inducting this procedure from $j = 2$ to $A.length$ to end up with a sorted array $A[1..n]$.

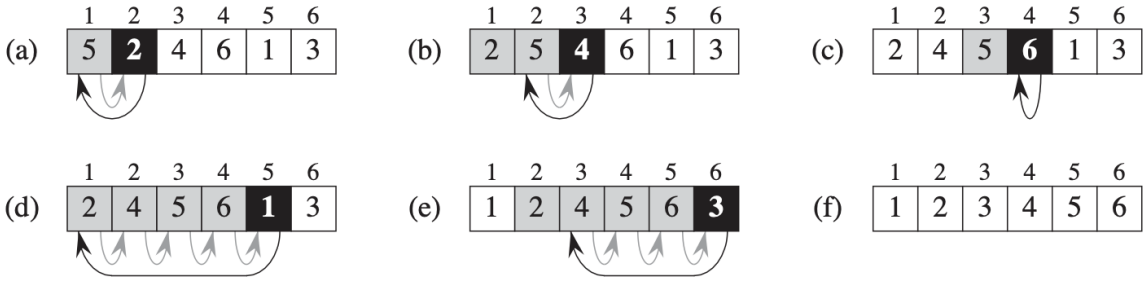

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

1.6 Assumptions

Before we can formally do a runtime analysis on insertion sort, we need to make a few assumptions. There are several operations involved in insertion sort such as comparing two

numbers, updating indicies with arithmetic operations, accessing value from an array. We have to follow a model of computation in order to make an assumption how these operations are done. For instance, we assume that accessing an element from an array takes a constant time.

In this course, for most of the time we will use ***random-access machine (RAM)*** model of computation. Under this model of computation, we assume that the following instructions take constant time (borrowed from page 24 of CLRS):

- Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- Data movement: load, store, copy
- Flow control: conditional and unconditional branch, subroutine call, return

1.7 Formal description of Insertion Sort

Formally, we define insertion sort with a pseudocode below:

- Input: A sequence of n numbers $\langle a_1, a_2, a_3, ..., a_n \rangle$.
- Output: A permutation or reordering $\langle a_1^{'}, a_2^{'}, a_3^{'}, ..., a_n^{'} \rangle$ of the input sequence such that $a_1^{'} \leq a_2^{'} \leq a_3^{'} \leq ... \leq a_n^{'}$.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

1.8 Proof of Correctness

Once we sketch out an algorithm, it is important to show that the algorithm works or the algorithms gives the correct output for the given input. We must find a way to provide the proof of correctness. There are several techniques to write a proof. *(A few notes on writing proofs has been posted on the site).* One may use proof by induction, proof by contradiction, proof by contrapositive, etc.

In order to show the correctness for insertion sort, we will use "loop invariants". "Invariant" essentially refers to a mathematical object which remains unchanged while going over a transformation. A very simple example to such invariant would be a triangle being rotated in a $\mathbb{R}^2$ space.

Now, one can guess what a "loop invariant" might be. Informally, a loop invariant is a statement or a condition that remains unchanged over iterations.

Lemma 1. At the start of each iteration of the outer for loop, the subarray $A[1, 2, ..., j-1]$ has the original elements in subarray $A[1, 2, ..., j-1]$, but in sorted order.

Notice the subarrays in figure 2.2 above for different value of j. For instance, when $j = 4$, the key is 6 at index 4, and the subarray we are interested in is, $A[1, 2, ..., j-1]$, which translates to $\langle 2, 4, 5\rangle$ subarray, which is sorted here. However, these elements, {2, 4, 5}, were not in sorted order previously, but they were placed within the same subarray. For instacen, when $j = 2$, subarray was $\langle 5, 2, 4\rangle$, when $j = 3$, subarray was $\langle 2, 5, 4\rangle$.

The loop invariant is the sorted subarray before going into the next iteration. If we can maintain this condition till the loop terminates, we must get a sorted array.

In order to use "loop invariants" for the proof of correctness, we need to prove the following three conditions:
Initialization: The loop invariant is true prior to the first iteration in of the loop.
Maintenance: If it is true before the i-th iteration of the loop, it remains true before the $(i + 1)$-th iteration of the loop.
Termination: When the loop terminates, the invariant gives us a useful property that helps us show the algorithm is correct.
One can notice a parallel between the "loop variant" and proof by induction.

- In a proof by induction, you have to establish the base case holds. Similarly, with "loop variant", we need to check if the "invariant" holds before the iteration.
- Showing that the invariant holds from iteration to iteration is similar to the inductive step.
- With mathematical induction, inductive step runs infinitely, while in "loop variant", the loop terminates. Therefore, the induction stops when the loop terminates.

For insertion sort:

Initialization: In the pseudocode, the loop starts at $j = 2$. Therefore, before the loop starts, the subarray $A[1, ..., j-1]$ will be $A[1]$, which is sorted trivially.
Maintenance: Let's assume $A[1, ..., j-1]$ is sorted. In pseudocode for insertion sort above, lines $(4-7)$ is for finding the proper place to insert $A[j]$, and line 8 does the proper insertion such that $A[1, ..., j]$ is sorted. Therefore, $A[1, ..., j]$ is sorted with the original elements from subarray $A[1, ..., j]$. Note that within the **while** loop, line 6 is shifting/moving $A[i]$ to right every time $i > 0$ and $A[i] > key$. You can think of this step as making a space to insert the *key*. Once a proper insertion is done to have a sorted subarray $A[1, ..., j]$, j is incremented to maintain the invariant. The loop invariant was sorted $A[1, ..., j-1]$ for $j-1$, and now it's sorted $A[1, ..., j]$ for j, ready for the next iteration.
Termination: Now, let's look at what happens when the loop terminates. The loop terminates when $j > A.length$ and $A.length = n$, which happens when $j = n+1$. As per the maintenance step, we can find that for $j = n+1$, we will have $A[1, ..., j-1] = A[1, ..., n+1-1] = A[1, ..., n]$ sorted as that is the loop invariant. Therefore, $A[1, ..., n]$ is sorted, which is the required goal when the loop terminates.

1.9 Runtime Analysis

1.9.1 Running Time

Let's calculate the number of steps involved in insertion sort. We will used the RAM model where we will assume steps such as comparison, addition, accessing values from a list are going to take a constant time. These constant values will be different for different operations.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

In outer **for** loop, to iterate from $j = 2$ to $n+1$, it will take n times for c_1 cost for each iteration. Therefore, each statement within the **for** loop will run $n-1$ times (line 2, 4, and 8). Let's assume, in **while** loop, the test condition will run t_j, which is based on the value of $i = j-1$. The **while** loop will iterate from $j = 2$ to n inclusive. Therefore, line 6 and 7 will run for $(t_j - 1)$ times for each j.

Putting everything together:

$$T(n) = c_1(n) + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

1.9.2 Best, worst, and average case analysis

Considering the fact that the runtime is based on the size of input array, n, the higher the value of n, the higher it will take to sort. Meanwhile, for a fixed size of n, there will be a best case and there will be a worst case.

The **best case** will to be sort a sequence which is already sorted. In that case, the **for** loop will run n times. However, the **while** loop will not have its body run as $A[i] > key$ will be **false** every time, as any number at j index will be larger than the number at $j-1$.

The **worst case** will be to sort a sequence which is reverse sorted, such as $\langle 5, 4, 3, 2, 1\rangle$. In this case, $t_j = j$. For instance, when $j = 2$, the while loop will do the loop condition check 2 times, and for $j = 3$, it will do the while loop check 3 times and so on. I will leave it as an exercise to prove $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$.

$$T(n) = c_1(n) + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

which can be rewritten as $T(n) = an^2 + bn + c$, implying that the runtime for insertion sort is a *quadratic* function of n.

For an **average case**, we need to consider the probability distribution of the permutations of the input sequence. If we consider the case where any *key*, $A[i]$, is less than the values in subarray $A[1, ..., i-1]$ and greater the rest of the values, the inner **while** loop has to run run $i/2$ times to find the proper index to drop the *key*, which implies that $t_i \approx i/2$. This will still lead $T(n)$ to be a quadratic function of n.

However, in this course, we will mostly be interested in the worst-case running time for any input of size n.

1.10 Class Exercise

1. Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41\rangle$.

2. Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of non-decreasing order.

3. How can we modify almost any algorithm to have a good best-case running time?