❖ Foundations 1: Learning the Language

Biologists study living organisms, psychologists study the human mind, and physicists study the laws of the universe. What exactly is it that a computer scientist does? The name is a bit misleading, as our study is not primarily about computers. That is something the engineers work on. A computer scientist is interested in studying **problems**.

When computer scientists talk about a "problem", we are usually dealing with a generalized notion of a relation between inputs and outputs. Consider the example of the problem of sorting a sequence of numbers in increasing order.

Problem (SORTING)**.**

Input: A sequence of n numbers $\mathbf{a} = \langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) of the numbers $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Given a problem, there are many ways to go from the input sequence to the output sequence. We will call an algorithm a procedural sequence of steps that solves a problem. When an algorithm satisfies two critical properties, we will call it a **correct algorithm**. Consider the following two examples of incorrect algorithms to think about what these properties may be.

Algorithm (HOPIUMSORT)**.**

1. If **a** is sorted, return **a**.
2. Else: type HOPIUM in the chat, then return to step 1. (HOPIUM is an emote that means "irrational or unwarranted optimism)

Algorithm (BOGOSORT*)**.**

1. Write down all the numbers in **a** on index cards.
2. Throw all the cards in the air and pick them up.
3. Write down the numbers on the cards in the order that you picked them up in a list, and return that list.

Question 1. Why would we not call HOPIUMSORT a good algorithm? Why would we not call BOGOSORT* a good algorithm? Given these problems, what properties might we demand to say that an algorithm is **correct**?

1.1 Analyzing Algorithms

For any given problem, there is a wide range of **correct algorithms** that can solve them. How do we choose which is better than another? An important answer to this question is that we want a fast algorithm.

Let's take a look at our first sorting algorithm.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        current = arr[i]
        # Insert current into the sorted subarray arr[0:i]
        j = i - 1
        while j > 0 and arr[j] > current:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = current
    return max
```

In this class, we will use what is called the **random-access machine (RAM) model** as our model of computation. In the RAM model we assume that,

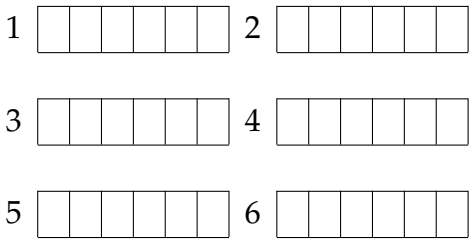
- instructions execute one after another,
- each instruction (addition, subtraction, multiplication, division, remainder, floor, ceiling) and data movement (load, store, copy), and control (condition/unconditional branch) takes the same amount of time.

In a sense, this is the simplest abstraction of a computer that is known to be a good predictor of the performance on a real device, even if it is using an architecture that is more sophisticated than the RAM model. Furthermore, we will see that it can be nontrivial to analyze the running time of algorithms even in such a simplified model. After all, we will be spending a whole quarter working on it!

Question 2. Suppose we call `insertion_sort` on the array

$$[4, 6, 1, 2, 5, 3]. \tag{1}$$

Write down the state of the array at the start of each iteration of the for loop.



Question 3. At each iteration, what can we say about the subarray `arr[0:i]`? (Remember, indexing in Python includes the start point, and doesn't include the end point)

1.1.1 Correctness

One way to formally prove that an algorithm that **loops** like `insertion_sort` works correctly is by using a **loop invariant**. What we answered in the above problem is the loop invariant for this algorithm. To prove this, we will use a very similar procedure to an inductive proof. We need to show that the following three things are true:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** The loop terminates, and when it terminates, the invariant - usually along with the reason that the loop terminated - gives us a useful property that helps show that the algorithm is correct.

Question 4. Show that the above three properties hold for `insertion_sort` in your own words.

- **Initialization:**

- **Maintenance:**

- **Termination:**

1.1.2 Efficiency

Correctness is just the first step in finding a good algorithm. As we will see, there can be many correct algorithms for the same problem, but many will have varying degrees of efficiency.

One important property we will be using to measure how good an algorithm is is by its running time. In the RAM model, we assume that each primitive instruction takes the same amount of time. Given this fact, the component of the algorithm that affects the running time is the size of the input. As such, we want to count the number of instructions that will be executed given an input of "size n". We will usually denote the running time of an algorithm on an input of size n by $T(n)$.

```
1  def insertion_sort(arr):
2      n = len(arr)
3      for i in range(1, n):
4          current = arr[i]
5          # Insert current into the sorted subarray arr[0:i]
6          j = i - 1
7          while j > 0 and arr[j] > current:
8              arr[j + 1] = arr[j]
9              j = j - 1
10         arr[j + 1] = current
11     return max
```

Question 5. On the right of each line of the function, write down the number of times that lines 2-6 and 10-11 will be executed.

Question 6. If the input array is sorted, how many times will lines 7-9 be executed?

Question 7. If the input array is in reverse order, how many times will lines 6-8 be executed?

Question 8. Write down an expression for the **worst case running time** $T(n)$ of insertion sort given our analysis from the previous page.

Note that we performed a **worst case analysis** for our algorithm here. In general, we will be interested in analyzing the **worst-case running time** of the algorithm over all possible inputs of size n. For some algorithms we will instead use **average case analysis** where we assume some distribution of inputs.

In summary, for any algorithm that we design to solve a particular problem, we want to make sure that

1. it is correct, and
2. how efficient it is.

1.2 Characterizing Running Times

We ended the previous section by giving a worst case bound on the number of operations that insertion sort must perform.

When we compare an input that has 6 elements vs. an input that has 1000 elements, we know that the lower order terms will become irrelevant. Furthermore, the coefficient of the n^2 terms is going to be the same for both cases, so we will discard that as well. In this case, we are going to say that asymptotically (as n increases) insertion sort has a running time of $\Theta(n)$ (read big-theta of n, or just theta of n).

We will be using a tool to classify functions as a way to categorize the different running times of our algorithms. The following language will be used to state precisely notions like

- $f(n) = \frac{3}{78}n - 6000$ grows faster than $g(n) = 30 \log n + 30$
- $a(n) = 7n^3 + 2n^2 + 10000$ grows slower than $b(n) = 100 \cdot 2^n + \log n$

1.2.1 O-notation (Informal)

We will use O-notation to characterize an *upper bound* on the asymptotic behavior of a function. In other words, we use it to say that our function grows *no faster* than a certain rate, based on the highest order term.

For two function f and g, we say $f = O(g)$ when the growth rate of f is at most ($\leq$) the growth rate of g.

Question 9. What is an asymptotic upper bound for the function $7n^3 + 100n^2 - 20n + 6$?

1.2.2 Ω-notation (Informal)

We will use Ω-notation to characterize a *lower bound* on the asymptotic behavior of a function. This means that the growth rate of the function is *at least* a certain rate, based on the highest order term.

For two function f and g, we say $f = \Omega(g)$ when the growth rate of f is at least ($\geq$) the growth rate of g.

Question 10. What is an asymptotic lower bound for the function $7n^3 + 100n^2 - 20n + 6$?

1.2.3 Θ-notation (Informal)

Finally, we will use Θ-notation to characterize a *tight bound* on the asymptotic behavior of a function. This means that the growth rate of the function is *precisely* at a certain rate, based on the highest order term. We will see what this means more formally later, but a way to think about it is saying that the growth rate of a function can be upper bounded and lower bounded to within a constant factor.

For two function f and g, we say $f = \Theta(g)$ when the growth rate of f is equal to the growth rate of g.

If a function is $O(f(n))$ and $\Omega(f(n))$ for the same function f, then we have shown that the function if $\Theta(f(n))$.

Question 11. Given the previous two problems, how can we characterize the asymptotic behavior of $7n^3 + 100n^2 - 20n + 6$?

Question 12. Let $T(n)$ be the number of operations required for an instance of insertion sort with an input array of size n.

- Use asymptotic notation to characterize the upper bound on the worst case running time.

- Use asymptotic notation to characterize the lower bound on the worst case running time.

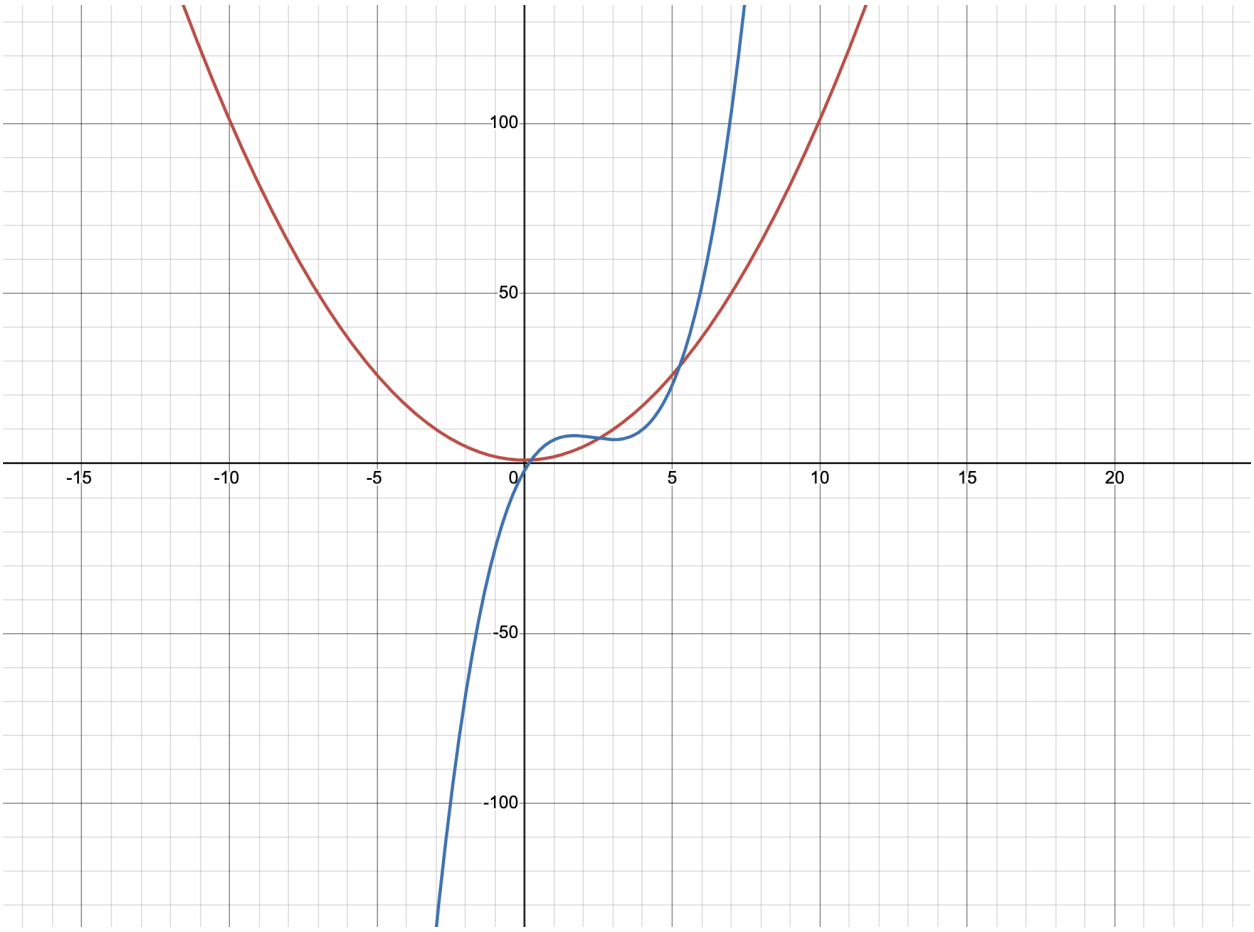
- Can we combine the above two to make a more precise statement about the running time of insertion sort?

❖ Foundations 2: Asymptotic notation and Mathematical Preliminaries

2.1 Formal Definitions

Let's see how we might want to formalize the ideas from above. Asymptotic notation is interested in the long term behavior of a function. In other words, we don't really care what happens early on in a function, as long as at some point the relation becomes clear.

Example 2.1. Let $f(n) = n^2$ and $g(n) = n^3 + 2n^2 - 3$.



The functions intertwine in the range from 0 to 5, but after 6 (more precisely around 5.23), we know that f will never be higher than g. This is the notion we want to capture when we say $f = O(g)$ (the rate of growth of f is $\leq$ the rate of growth of g).

2.1.1 O-notation

Asymptotic notation is defined using sets.

Definition 2.2 (O-notation)**.** Let g be a function. Then, we define $O(g)$ to be the *set of functions*

$$O(g) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

Even though it is a set, instead of using the notation

$$f(n) \in O(g(n)), \tag{2}$$

the standard literature will use

$$f(n) = O(g(n)). \tag{3}$$

These two mean the same thing so we will use them interchangeably.

Question 13. Show that $4n^2 + 100n + 500 = O(n^2)$. (Hint: you need to find positive constants c and n_0 such that the $f(n) \leq cg(n)$ for all $n \geq n_0$.)

- What is $f(n)$?
- What is $g(n)$?

2.1.2 Ω-notation

Definition 2.3 (Ω-notation)**.** Let g be a function. Then, we define $\Omega(g)$ to be the *set of functions*

$$O(g) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$$

Question 14. Show that $17n \log n = \Omega(n)$.

2.1.3 Θ-notation

Remember, O and Ω give upper and lower bounds on the growth rates of the functions respectively. When we are able to have the same upper and lower bounds reach the same family of functions, we can make a stronger statement on the behavior of the function.

Definition 2.4 (Θ-notation)**.** Let g be a function. Then, we define $\Theta(g)$ to be the *set of functions*

$$O(g) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

Alternatively, we can use the following definition. Given two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Question 15. Show that $3n^3 + 50n^2$ does not belong to the set $O(n^2)$.

Question 16. Show that $n^2 \log n$ does not belong to the set $\Omega(n^3)$.

2.1.4 o-notation

Both O and Ω give us bounds that may or may not be asymptotically tight. For example, The bound $3n^2 = O(n^2)$ is asymptotically tight, but $3 \log n = O(\log^4 n)$ is not. We use o-notation to denote an upper bound that is **not** asymptotically tight.

Definition 2.5 (o-notation)**.** Let f and g be two functions. Then, we say that $f(n) = o(g(n))$ iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Question 17. Show that $3 \ln(n) = o(n)$.

2.2 Mathematical Preliminaries

2.2.1 Sums

We will often use summation notation:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b). \tag{4}$$

Question 18. Write down the following expression using summation notation:

$$3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 \tag{5}$$

Sometimes we want to sum over the values in a set $S = \{x_1, x_2, \ldots, x_n\}$. In that case, we may use the following notation

$$\sum_{x \in S} f(x) = f(x_1) + f(x_2) + \cdots + f(x_n). \tag{6}$$

Question 19. Let $S = \{3, 4, 7, 12, 15\}$. Write down the following summation explicitly:

$$\sum_{x \in S} \log(x+1) \tag{7}$$

A special case of sums that we will be very interested in is the **geometric sum**:

$$\sum_{i=0}^{n} c^i = c^0 + c^1 + \cdots + c^n \tag{8}$$

where c is some constant number.

In your homework, we will ask you to prove the following identity:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^n}{1 - a}. \tag{9}$$

2.2.2 Logarithms

In computer science, we are very interested in exponentially growing functions. In general, when we are interested in a class of functions we are also interested in its **inverse** function as well. The **logarithm** is the inverse function of exponential functions.

$$\log_b x = y \text{ iff } x = b^y. \tag{10}$$

Here are some useful properties of the logarithm that we will use throughout the course.

1. $\log_b 1 = 0$.
2. $\log_b b^a = a$.
3. $\log_b(xy) = \log_b(x) + \log_b(y)$.
4. $\log_b x^a = a \log_b x$.
5. $x^{\log_b y} = y^{\log_b x}$.
6. $x^{\log_x b} = \frac{1}{\log_b x}$.
7. $\log_a x = \frac{\log_b a}{\log_b x}$.
8. $\log_a x = (\log_b x)(\log_a b)$.

Question 20. Prove property 2 from the above list.

2.2.3 Floors and Ceilings

Let x be some real number.

- Floor: $\lfloor x \rfloor$ is the largest integer that is less than or equal to x.
- Ceiling: $\lceil x \rceil$ is the smallest integer that is greater than or equal to x.

Question 21. What is

- $\lfloor 3.5 \rfloor$?
- $\lfloor -7.5 \rfloor$?
- $\lceil -\pi \rceil$?
- $\lceil 7.5 \rceil$?

2.2.4 Factorials

Question 22. How many ways can I reorder the numbers 4, 6, 8?

The expression $n!$ (read n-factorial) is defined for any nonnegative integer:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n. \tag{11}$$

We use this expression as the way to calculate the number of permutations there are for n distinct elements.

2.2.5 Combinations

Question 23. How many ways can we choose 2 items from a bag of 5 items?

We also have some notation to represent this idea. Let $\binom{n}{k}$ (read n choose k) be the number of ways to choose k items from n items. We have the following closed form expression:

$$\binom{n}{k} := \frac{n!}{(n-k)!k!}. \tag{12}$$

2.2.6 Probability

When we discuss probability, we will always be referring to probabilities with respect to some **sample space** S. A subset of the sample space will be referred to as an **event**.

Example 2.6. Consider the game of flipping two coins. If you get two of the same face in a row, you will win 100.

- What is the sample space for this game?
- What is the event "winning the game"?

Given our sample space, we define the probability function $\Pr(\cdot)$ as a function that maps an event to a real number with the following properties:

1. $\Pr(\emptyset) = 0$
2. $\Pr(S) = 1$
3. For every event A, $0 \leq \Pr(A) \leq 1$.
4. If A and B are events such that $A \cap B = \emptyset$, then $P(A \cup B) = \Pr(A) + \Pr(B)$.

Question 24. Suppose that the game we are playing above uses two fair coins. Then, $\Pr(HH) = \Pr(HT) = \Pr(TH) = \Pr(TT) = \frac{1}{4}$. What is the probability of winning the game?

A **random variable** is defined as a function from a set of outcomes in a sample space to real numbers. Let X be a random variable representing your payout from the above game. We can express X as a function of the outcomes:

$$X(s) = \begin{cases} 100 \text{ if } s = HH \text{ or } s = TT. \\ 0 \text{ otherwise.} \end{cases} \tag{13}$$

Given a random variable, we can find what the **expected value** $\mathbb{E}[X]$ of the random variable X is. This is a generalization of the idea of the **average**. Suppose that X can take the values $V = \{x_1, \ldots, x_n\}$. Then,

$$\mathbb{E}[X] := \sum_{x \in V} x \cdot P(X = x). \tag{14}$$

Question 25. Let's play a similar game with a six sided dice. If you roll an i, I will give you $2i$ cookies. Assuming the dice is fair, what is the expected number of cookies we will get from playing one round of this game?

Linearity of expectation: For any two random variables X and Y, we have

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]. \tag{15}$$

For any two events A and B, we will say that they are **independent** if and only if

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B). \tag{16}$$

Question 26. Suppose we have two fair coins. Define the two events

- A = coin 1 is Heads: $\{HH, HT\}$,
- B = coin 2 is Tails: $\{HT, TT\}$.

Are these two events independent?

We will say a collection of n events $C = \{A_1, \ldots, A_n\}$ is **mutually independent** (or sometimes just independent) if for every subset $\{A_{i1}, \ldots, A_{ik}\}$ of C,

$$\Pr(A_{i1} \cap \cdots \cap A_{ik}) = \Pr(A_{i1}) \cdot \Pr(A_{i2}) \cdots \Pr(A_{ik}) \tag{17}$$

Question 27. Suppose we flip 10 fair coins and that the outcomes are all independent. What is the probability that we see the sequence $HTHHT\ TTHTH$?