Week 1 work: Sept. 4 — Sept. 12
12-hour week
Obligatory problems are marked with [**]
1. **[Gates for universal classical computation.]**

(a) Show that any Boolean function \( f : \{0,1\}^n \rightarrow \{0,1\} \) can be computed by a classical Boolean circuit using the following set of logic gates: 2-bit AND, 2-bit OR, and NOT. (Hint: look up *DNF formula.*)

(b) Show that any Boolean function \( f : \{0,1\}^n \rightarrow \{0,1\} \) can be computed by a classical Boolean circuit using the following single logic gate: 2-bit NAND. Also show this for the following single logic gate: 2-bit NOR.

(c) Show that there are infinitely many Boolean functions \( f : \{0,1\}^n \rightarrow \{0,1\} \) that cannot be computed by a classical Boolean circuit using the following set of logic gates: 2-bit AND, 2-bit OR.

(d) Show that there are infinitely many Boolean functions \( f : \{0,1\}^n \rightarrow \{0,1\} \) that cannot be computed by a classical Boolean circuit using the following set of logic gates: 2-bit XOR, and NOT.
2. [Reviewing big-O.] Review “big-O” notation, e.g., by reading this, or reading the first part of Chapter 6 here, or by watching this.

I will use sometimes one more piece of notation: “O-tilde”, or “soft big-O notation”. Basically, \( \widetilde{O}(g(n)) \) means “big-O of \( g(n) \), ignoring logarithmic factors”. More formally, we say that \( f(n) = \widetilde{O}(g(n)) \) if \( f(n) = O(g(n) \cdot (\log g(n))^c) \) for some constant \( c \). Some exercises for you:

(a) Is \( 10n^2 \log n = \widetilde{O}(n^2) \)?
(b) Is \( 100n^2(\log n)^3 = \widetilde{O}(n^2) \)?
(c) Is \( 5(\log n)^2 = \widetilde{O}(1) \)?
(d) Is \( n^3 = \widetilde{O}(n^2) \)?
(e) Is \( 3^n = O(2^n) \)?
(f) Is \( 3^n = \widetilde{O}(2^n) \)?
(g) Is \( 3^n \cdot n^2 = O(3^n) \)?
(h) Is \( 3^n \cdot n^2 = \widetilde{O}(3^n) \)?
(i) Explain why a list of \( n \) numbers can be sorted in \( \widetilde{O}(n) \) time.
3. [Computational arithmetic.]

(a) Watch this lecture on how to multiply two \( n \)-bit numbers in \( \tilde{O}(n) \) steps using the Fast Fourier Transform. (Budget 1 hour at 1.25× or 1.5× speed.)

(b) Consider the “long division algorithm” for integers that you learn in grade school. Given two numbers \( C \) and \( D \), it outputs the (integer) quotient \( Q = \lfloor C/D \rfloor \) and the remainder \( R = C \mod D \). Argue that if \( C \) and \( D \) are both at most \( n \) digits, then this algorithm will compute \( Q \) and \( R \) in at most \( \tilde{O}(n^2) \) operations. 

(Remark: in fact, there’s a sophisticated way to efficiently reduce integer division to integer multiplication, meaning that integer division can actually be done in \( \tilde{O}(n) \) operations. The infamous “Pentium bug” was due to messing up this reduction.)

(c) [**] Consider the following task: Given positive integers \( B \) and \( C \), compute the integer \( B^C \). Show that this task is not solvable “in \( P \)”; that is, there is no algorithm that can do this in \( \tilde{O}(n^{\text{constant}}) \) operations when \( B \) and \( C \) are \( n \)-bit numbers. (Hint.)

(d) [**] Consider the following task: Given positive integers \( B, C, \) and \( D \), compute the integer \( B^C \mod D \). This is called the modular exponentiation problem. Show that this task is solvable “in \( P \)”. If \( B, C, \) and \( D \) are all \( n \)-bit numbers, show that it can be done in \( \tilde{O}(n^3) \) steps. (In fact, it can be done in \( \tilde{O}(n^2) \) steps using the sophisticated multiplication and division algorithms.)

(Hint: One key fact to use is

\[
P \cdot Q \mod D = (P \mod D) \cdot (Q \mod D) \mod D.
\]

Given this, first think about computing \( B \mod D, B^2 \mod D, B^4 \mod D, B^8 \mod D, B^{16} \mod D, \) etc. If \( C \) happens to be a power of 2, you should be in good shape. What should you do if \( C \) is, say, 24? What should you do if \( C \) is (when represented in base 2) 1010101010101010?)

---

\( ^1 \) Some evidence...
4. [Simulating a biased coin.] The usual way to obtain a model of probabilistic computation is to take a standard model of deterministic computation (e.g., Turing Machines, Boolean circuits, your favorite programming language) and add a new “FLIP\(_{1/2}\)” operation, which by definition returns 0 with probability 1/2 and returns 1 with probability 1/2.

A more liberal augmentation would be to allow the “FLIP\(_p\)” operation for any rational value 0 < \(p\) < 1, which by definition returns 0 with probability 1 − \(p\) and returns 1 with probability \(p\). This problem is about exploring the difference between the two models.

(a) In one sense, general FLIP\(_p\) operations are more powerful than FLIP\(_{1/2}\) operations. Show that if you only get FLIP\(_{1/2}\) operations, it’s impossible to exactly simulate a FLIP\(_{1/3}\) gate.

(b) [**] However, in another sense, FLIP\(_p\) operations are not fundamentally more powerful than FLIP\(_{1/2}\) operations. Writing in pseudocode, prove that for any \(\epsilon > 0\), there is a simple subroutine using only deterministic computation and FLIP\(_{1/2}\) operations that almost exactly simulates a FLIP\(_{1/3}\) operation, in the following sense: Your subroutine should return a value \(r \in \{0, 1, \text{FAIL}\}\), and it should have the following two properties:
   (i) \(\Pr[r = \text{FAIL}] \leq \epsilon\); and, (ii) \(\Pr[r = 1 \mid r \neq \text{FAIL}] = 1/3\) exactly.

   (Remark: This problem is doable for any rational value of \(p\), not just 1/3; but I expect that once you solve it for 1/3, you’ll get the idea of how to do it for any \(p\).)

(c) Implement and test your solution in your favorite programming language, with \(\epsilon = 2^{-500}\).

(d) (Requires a bit of sophistication in Theoretical Computer Science thinking.) Suppose that you augment deterministic computation by allowing a FLIP\(_p\) operation for any real 0 < \(p\) < 1. Further, the algorithm designer only needs to mathematically specify each \(p\) used; the algorithm itself doesn’t have to “calculate” \(p\) or anything. (Think, e.g., of FLIP\(_{1/\pi}\) operations.) You might imagine the algorithm is given a “magic coin” with bias \(p\), for any \(p\) of the algorithm designer’s choosing. Does this give fundamentally increased power over deterministic computation?
5. [Dealing with error in randomized computation.] Suppose you are trying to write a computer program \( C \) to compute a certain Boolean function \( f : \{0, 1\}^n \to \{0, 1\} \), mapping \( n \) bits to 1 bit. (For example, perhaps \( f \) specifies that \( f(x) = 1 \) if and only if \( x \) represents a prime number written in base 2.) If \( C \) is a deterministic algorithm, then there is an obvious definition for "\( C \) successfully computes \( f \)"; namely, it should be that \( C(x) = f(x) \) for all inputs \( x \in \{0, 1\}^n \). But what if \( C \) is a probabilistic algorithm?

The best thing is if \( C \) is a zero-error algorithm for \( f \), with failure probability \( p \). This means:

- on every input \( x \), the output of \( C(x) \) is either \( f(x) \) or is "?"
- on every input \( x \) we have \( \Pr[C(x) =?] \leq p \)

Important note: The second condition is not about what happens for a random input \( x \). Instead, it demands that for every input \( x \) the probability of failure is at most \( p \), where the probability is only over the internal "coin flips" of \( C \).

(a) [**] If you have a zero-error algorithm \( C \) for \( f \) with failure probability 90% (quite high!), show how to convert it to a zero-error algorithm \( C' \) for \( f \) with failure probability at most \( 2^{-500} \). The "slowdown" should only be a factor of a few thousand.

(b) [**] Alternatively, show how to convert \( C \) to an algorithm \( C'' \) for \( f \) which: (i) always outputs the correct answer, meaning \( C''(x) = f(x) \); (ii) has expected running time only a few powers of 2 worse than that of \( C \). (Hint: look up the mean of a geometric random variable.)

The second best thing is if \( C \) is a one-sided error algorithm for \( f \), with failure probability \( p \). There are two kinds of such algorithms, "no-false-positives" and "no-false-negatives". For simplicity, let’s just consider “no false-negatives” (the other case is symmetric); this means:

- on every input \( x \), the output \( C(x) \) is either 0 or 1
- on every input \( x \) such that \( f(x) = 1 \), the output \( C(x) \) is also 1
- on every input \( x \) such that \( f(x) = 0 \), we have \( \Pr[C(x) = 1] \leq p \)

(c) [**] If you have a no-false-negatives algorithm \( C \) for \( f \) with failure probability 90% (quite high!), show how to convert it to a no-false-negatives algorithm \( C' \) for \( f \) with failure probability at most \( 2^{-500} \). The “slowdown” should only be a factor of a few thousand.

The third best thing (in fact, the worst thing, but it's still not so bad) is if \( C \) is a two-sided error algorithm for \( f \), with failure probability \( p \). This means:

- on every input \( x \), the output \( C(x) \) is either 0 or 1
- on every input \( x \) we have \( \Pr[C(x) \neq f(x)] \leq p \)

Remark: It is actually very very rare in practice for a probabilistic algorithm to have two-sided error; in almost every natural case, an algorithm you design will have one-sided error at worst.

(d) If you have a two-sided error algorithm \( C \) for \( f \) with failure probability 40%, show how to convert it to a two-sided error algorithm \( C' \) for \( f \) with failure probability at most \( 2^{-500} \). The “slowdown” should only be a factor of a few dozen thousand. (Hint: look up the Chernoff bound.)
6. [CMU Probabilistic Experience.]

(a) Play around with the IBM Q Experience.

(b) [**] Write a “coin-flipping experience” program in your favorite programming language. Your program should support a fixed number of coins $n$ (you choice; say, $5 \leq n \leq 10$), each of which can be showing 0 (Heads) or 1 (Tails). It is assumed that all coins are initialized to be 0/Heads. The input to your program should be the description of a “circuit” (in any convenient format of your choice; e.g., a text file). A circuit is just an arbitrary-length sequence of operations from the following set:

- **Flip** $i$ (randomly set coin $i$ to 0 or 1 with probability $1/2$ each)
- **Not** $i$ (turn over the $i$th coin; i.e., deterministically reverse its 0/1 status)
- **CNot** $i j$ (if coin $i$ is 1 (Tails) then do a Not on coin $j$, else do nothing)
- **CSwap** $i j k$ (if coin $i$ is 1 (Tails) then swap the values of coins $j$ and $k$)

In the above, $i,j,k$ stand for distinct coin numbers between 1 and $n$.

If you like, you can also implement the following operations:

- **CCNot** $i j k$ (if coins $i$ and $j$ are both 1 then do ‘Not $k$’, else do nothing)
- **GenFlip** $i p$ (set coin $i$ to 0 with probability $1 - p$, to 1 with probability $p$)
- **Gen1Bit** $i p q$ (if coin $i$ is 0 then make it 1 with probability $p$, else if coin $i$ is 1 then make it 0 with probability $q$)

Given the input circuit description, your program should use (pseudo)randomness to simulate one run of the circuit and output the resulting final outcome of the coins (a length-$n$ bitstring). (You should test your program with multiple runs to make sure it works!)

---

2“Bonus points” if you do it in Scratch.
7. [Abandoning realism.]

(a) [**] Following on from the CMU Probabilistic Experience problem, make a new version of your program that takes as input the description of a circuit, and calculates the probabilities of each possible outcome. Your new program should output these probabilities as a column of $2^n$ numbers (adding up to 1). E.g., if $n = 5$ then the output should be

\begin{align*}
\Pr[\text{circuit would output 00000}] \\
\Pr[\text{circuit would output 00001}] \\
\Pr[\text{circuit would output 00010}] \\
\cdots \\
\Pr[\text{circuit would output 11111}]
\end{align*}

These numbers should be exactly calculated; they should not be obtained by simulating your previous programming and taking an empirical average.³ (Hint: it might help you if your favorite programming language has built-in support for matrix multiplication.)

(b) Upgrade your program so that instead of assuming all coins are initialized to 0, your program outputs one column of results for each of the $2^n$ possible initial settings of the coins. (Thus your program should be outputting a $2^n \times 2^n$ matrix, with rows and columns indexed by length-$n$ bitstrings, in which the entry in the $x$th column and $y$th row is the probability that the circuit outputs $y \in \{0, 1\}^n$ given that its input is initialized to $x \in \{0, 1\}^n$.)

³“Bonus points” if you implement GenFlip and Gen1Bit and then give the output answers symbolically as a polynomial functions of all the $p$'s and $q$'s.
8. [Miscellaneous.]

- Watch this video by 3B1B on the enormity of the number $2^{256}$.
- Read this survey by Pomerance on factoring.
- Watch this surprisingly accurate PBS video on the Many Worlds Interpretation.
- Send an email to the instructor (odonnell@cs.cmu.edu) saying hello, what year and program you’re in, what your interest in the course is, and one of the following: (i) interesting fact about yourself; (ii) your hometown; (iii) your favorite show.