CS4268, AY 2025-26 Sem 2
Lecturer: Divesh Aggarwal [1]

Tutorial Week 4

---

1 [Gates for universal classical computation.]

(a) Show that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ 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 $f : \{0,1\}^n \to \{0,1\}$ 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 $f : \{0,1\}^n \to \{0,1\}$ 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 $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a classical Boolean circuit using the following set of logic gates: 2-bit XOR, and NOT.

2 [Computational arithmetic.]

Consider the following task: Given positive integers B, C, and D, compute the integer $B^C \bmod D$. 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

$$P \cdot Q \bmod D = (P \bmod D) \cdot (Q \bmod D) \bmod D.$$

Given this, first think about computing $B \bmod D$, $B^2 \bmod D$, $B^4 \bmod D$, $B^8 \bmod D$, $B^{16} \bmod D$, 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?)

3 [Simulating a biased coin.]

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

(b) However, in another sense, FLIP_p operations are not fundamentally more powerful than $\text{FLIP}_{1/2}$ operations. Writing in pseudocode, prove that for any $\varepsilon > 0$, there is a simple subroutine using only deterministic computation and $\text{FLIP}_{1/2}$ operations that almost exactly simulates a $\text{FLIP}_{1/3}$ 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:

[1] Problems are taken from the homework sheets of Ryan O'Donnell's course in 2018; https://www.cs.cmu.edu/~odonnell/quantum18/

(i) $\Pr[r = \text{FAIL}] \leq \varepsilon$; and,

(ii) $\Pr[r = 1 \mid r \neq \text{FAIL}] = 1/3$ exactly.

(c) (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 $\text{FLIP}_{1/\pi}$ 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?

4 [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.

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.)

5 [Complex number exercises.]

Let $z = x + iy$ be a complex number, where x and y are real numbers and $i = \sqrt{-1}$.

Verify the following facts:

(a) $\mathrm{Re}(z) = \frac{1}{2}(z + z^*)$

(b) $\mathrm{Im}(z) = \frac{1}{2i}(z - z^*)$

(c) $(z + w)^* = z^* + w^*$

(d) $(zw)^* = z^* w^*$

(e) $|z| = \sqrt{zz^*}$

(f) $1/z = \frac{x}{x^2+y^2} - i\frac{y}{x^2+y^2}$

(g) If (r, θ) are the polar coordinates of z, then $r = \sqrt{zz^*}$ and $\theta = \tan^{-1}(\mathrm{Im}(z)/\mathrm{Re}(z))$

(h) Conversely, $z = re^{i\theta} = r(\cos\theta + i\sin\theta)$

(i) $1/z$ has polar coordinates $(1/r, -\theta)$

(j) z^* has polar coordinates $(r, -\theta)$

(k) if $|z| = 1$, then $1/z = z^*$

(l) If w has polar coordinates (R, ϕ), then zw has polar coordinates $(rR, \theta + \phi)$

(m) If z has polar coordinates (r, θ) then z^n has polar coordinates $(r^n, n\theta)$

(n) If $n \in \mathbb{Z}^+$, there are exactly n distinct complex numbers w satisfying $w^n = 1$

(o) Let ω_n be the primitive nth root of unity. Show that $\{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ are exactly the nth roots of unity

6 [Dirac notation and measurement exercises.]

Let $|\phi\rangle = 3\,|0\rangle - 5i\,|1\rangle$.

(a) What is $\langle\phi|\phi\rangle$?

(b) What number C should $|\phi\rangle$ be divided by to make it a normalized state?

(c) What are the possible outcomes and associated probabilities if $|\psi\rangle$ is measured in the $\{|0\rangle, |1\rangle\}$ basis?

(d) Same question as above for measuring in the $\{|+\rangle, |-\rangle\}$ basis.

(e) Verify that $\frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle$ and $\frac{1}{\sqrt{2}}|0\rangle - \frac{i}{\sqrt{2}}|1\rangle$ form an orthonormal basis for $\mathbb{C}^2$.

7 [Projectors and reflections.]

Let $|\psi\rangle$ and $|\phi\rangle$ be unit vectors in $\mathbb{R}^d$. Define $Q = |\phi\rangle\langle\psi|$.

(a) Explicitly work out the matrix Q in the case $|\psi\rangle = |0\rangle$ and $|\phi\rangle = |+\rangle$, and also in the opposite case.

(b) Practice hand-drawing the expression $|\phi\rangle\langle\psi|$.

(c) Fill in the blanks: The transformation Q maps $|\psi\rangle$ to ___, and maps every vector orthogonal to $|\psi\rangle$ to ___.

(d) Suppose now that $|\psi\rangle = |\phi\rangle$. Let $P = |\psi\rangle\langle\psi|$. Describe the transformation P.

(e) Let 1 denote the identity matrix. Describe the transformation $1 - 2P$ and prove that it is unitary.

(f) Show that the change-of-basis operator

$$U = |\phi_1\rangle\langle\psi_1| + \cdots + |\phi_d\rangle\langle\psi_d|$$

is unitary.

8 [Unitary matrices.]

Let $A \in \mathbb{C}^{d\times d}$ be a matrix that preserves lengths; that is, $\|A|\psi\rangle\| = \||\psi\rangle\|$ for all $|\psi\rangle \in \mathbb{C}^d$. Prove that A is unitary.

9 [Quantum Anti-Zeno Effect.]

Assume you have a single qubit that you know is in the state $|0\rangle$. You really wish to change its state to $|1\rangle$. You have the ability to build any measurement device, and use it as many times as you want. How can you (almost surely) get the qubit's state changed to $|1\rangle$?

10 [GCD.]

This problem is about the task of computing the GCD (greatest common divisor) of two input numbers, A and B. As usual, you should imagine these to be numbers to be, like, 2000 binary digits long. The grade school algorithm for GCD is: (i) find the prime factorizations of A and B; (ii) pick out all the common prime factors, and multiply them together. Of course, this algorithm is not actually possible to implement in physical reality for 2000-bit numbers (given known classical factoring algorithms). However, there is a physically possible algorithm (i.e., computing GCD is in "P"): it is called Euclid's Algorithm.

(a) (Warmup to Euclid's Algorithm — how he actually described it.) Show that if Q is a divisor of both A and B, then it's also a divisor of $A - B$. Conversely, show that if Q is a divisor of $A - B$ and B, then it's also a divisor of A. Conclude the rule

$$\gcd(A, B) = \gcd(A - B, B).$$

(b) Suppose you were computing $\gcd(A, 6)$, where $A = 6 \times 10^{500} + 4$. You would not want to do the subtraction rule 10^{500} times before getting to the swapping rule. But it should be obvious what subproblem you'll get down to after performing all those subtractions. Prove that the following is a correct algorithm for computing the GCD:

$$\text{Euclid}(A, B) : \begin{cases} \text{if } B = 0, \text{ return } A \\ \text{else return Euclid}(B, A \bmod B) \end{cases}$$

(c) When we execute $\text{Euclid}(A, B)$, it produces a descending chain of numbers; e.g., $\text{Euclid}(100, 18)$ produces

$$100,\ 18,\ 10,\ 8,\ 2.$$

Any three consecutive numbers in this chain are of the form $C, D, (C \bmod D)$. Prove that for any three consecutive numbers F_{t-1}, F_t, F_{t+1} in the chain, we have

$$F_{t+1} \leq \tfrac{1}{2} F_{t-1}.$$

Conclude that the total length of the chain is at most $\log_2 A + \log_2 B$.

11 [Perfect Powers.]

(a) Give pseudocode for an algorithm that takes as input a positive integer A and determines whether or not A is a perfect square. If it is, your algorithm should also determine the number B such that $B^2 = A$. If A is n binary digits long, your algorithm should take $O(nM(n))$ steps, where $M(n)$ is the number of steps required to multiply two numbers of at most n binary digits.

(b) Give pseudocode for an algorithm that takes as input a positive integer A and determines whether or not A is a perfect power (i.e., a perfect square, cube, fourth power, etc.). If it is, your algorithm should also determine numbers B and $C > 1$ such that $B^C = A$. When A is an n-bit number, justify that your algorithm takes at most $O(n^d)$ steps for some constant d.