CS 278: Computational Complexity Theory
Homework 2

Due: **October 24th, 2025**

Fall 2025

Instructions:

- Collaboration is allowed but solutions must be written independently. List collaborators and external resources.
- Write your solutions in LaTeX and submit a single PDF by email to `lijiechen@berkeley.edu` with subject "CS 278 - Homework 2 – [Your Name]".
- Name your file `CS278-HW2-[YourName].pdf`. **Deadline:** 11:59pm Pacific Time October 24th, 2025.
- Late submissions lose **10%** per day (e.g., three days late $\to 0.9^3$ of your score).
- This homework has 4 problems, each with multiple parts, totaling **160 points**. As in HW1, our course policy on homework aggregation applies unchanged.

1 Problem 1 (40 pts): A circuit lower bound for quadratic space

Show that languages decidable in quadratic space do not admit linear-size circuits.

Statement. Prove that $\mathrm{SPACE}[n^2] \not\subseteq \mathrm{SIZE}(O(n))$. In other words, there is a language $L \in \mathrm{SPACE}[n^2]$ such that every Boolean circuit family computing L on inputs of length n has size $\omega(n)$ for all sufficiently large n.

Name: ______________

2 Problem 2 (40 pts): NEXP $\subseteq$ coNEXP$/_{n+1}$

Statement. Prove that NEXP $\subseteq$ coNEXP$/_{n+1}$. In other words, for every language $L \in$ NEXP, prove that there exists an advice function $a(n) \in \{0,1\}^{n+1}$ such that there exists a coNEXP machine M that, for every $n \in \mathbb{N}$, given the correct advice $a(n)$, M computes L on all n-bit inputs.

3 Problem 3 (40 pts): Prove $\mathrm{CL} \subseteq$ ZPP

The class CL (catalytic logspace) allows an algorithm logarithmic *clean* space and polynomially many *dirty* bits that must be restored at the end. Show that every $L \in \mathrm{CL}$ has a zero-error expected-polynomial-time algorithm.

Definition of ZPP. The complexity class ZPP (Zero-error Probabilistic Polynomial time) consists of all languages L for which there exists a probabilistic polynomial-time Turing machine M such that:

1. For every input x, $M(x)$ outputs either 0, 1, or $\perp$ ("don't know").
2. If $x \in L$, then $\Pr[M(x) = 1] \geq \frac{1}{2}$ and $\Pr[M(x) = 0] = 0$.
3. If $x \notin L$, then $\Pr[M(x) = 0] \geq \frac{1}{2}$ and $\Pr[M(x) = 1] = 0$.
4. The expected running time of M on any input is polynomial.

Equivalently, ZPP is the class of languages that can be decided by Las Vegas algorithms: randomized algorithms that always give the correct answer when they terminate, and have polynomial expected running time.

Statement. Prove that $\mathrm{CL} \subseteq \mathrm{ZPP}$.

4 Problem 4 (40 pts): Prove that $\mathbf{TC}^1 \subseteq \mathrm{CL}$

In the class we sketch the high-level idea of the proof that $\mathrm{TC}^1 \subseteq \mathrm{CL}$. In this problem, you will complete the proof.

Statement. Prove that $\mathrm{TC}^1 \subseteq \mathrm{CL}$. You should give a complete register program for the TC^1 circuit and prove it's correctness.

Optional references and context

For accessible background on catalytic logspace and the toggling construction, see:

- Nathan Sheffield, *A quick-and-dirty intro to CL.*
- Ian Mertz, *Reusing Space: Techniques and Open Problems.*

(Optional) Solution placeholders

You may use the following headings for your writeup; remove them if not needed.

Solution to Problem 1

Goal. Prove that there exists a language $L \in \text{SPACE}[n^2]$ such that for every constant c, for all sufficiently large n, no Boolean circuit of size at most cn computes L on $\{0,1\}^n$; equivalently, $\text{SPACE}[n^2] \not\subseteq \text{SIZE}(O(n))$.

Proof. We will proceed in steps.

Step 1: Reduction to a hard function on $2\log n$ bits.

Given input $x \in \{0,1\}^n$. We are going to construct a function $f_n : \{0,1\}^{2\log n} \to \{0,1\}$ that is hard for circuits of size $O(n^{1.1})$. Our language L is going to be defined as $L = \{x \in \{0,1\}^n : f_n(x_{\leq 2\log n}) = 1\}$.

Note that if f_n is hard for circuits of size $n^{1.1}$, then L is hard for circuits of size $O(n)$.

Step 2: Enumerating and testing functions.

Fix $m = 2\log n$. We will construct a function $f : \{0,1\}^m \to \{0,1\}$ that is hard for circuits of size $c \cdot m$ for a suitable constant c.

Our $\text{SPACE}[n^2]$ algorithm does the following:

1. **Enumerate:** Iterate through all $2^{2^m} = 2^{n^2}$ possible truth tables for functions $f : \{0,1\}^m \to \{0,1\}$, in lexicographic order. Each truth table can be represented as a string of length $2^m = n^2$, which fits in n^2 space.

2. **Test hardness:** For each candidate function f, check whether f can be computed by some Boolean circuit of size at most $n^{1.1}$.

   To perform this test in $\text{SPACE}[n^2]$:

   - Enumerate all possible circuits C with at most $n^{1.1}$ gates over the basis $\{\wedge, \vee, \neg\}$ (or any complete basis).
   - For each circuit C, enumerate all $2^m = n^2$ possible inputs $x \in \{0,1\}^m$ and check whether $C(x) = f(x)$ for all x.
   - The number of circuits of size s is at most $s^{O(s)}$, so there are at most $(n^{1.1})^{O(n^{1.1})} = 2^{O(n^{1.1}\log n)}$ circuits to check.
   - Each circuit evaluation takes polynomial space (in fact, logarithmic space suffices).
   - We can enumerate circuits using a counter of size $O(n^{1.1}\log n)$ bits, which fits in n^2 space for large n.

3. **Output:** Return the lexicographically first function f that is *not* computable by any circuit of size $n^{1.1}$.

Step 3: Existence of a hard function.

By a counting argument, such a hard function must exist. The number of Boolean functions on $m = 2\log n$ bits is $2^{2^m} = 2^{n^2}$. The number of circuits of size at most $n^{1.1}$ is at most $2^{O(n^{1.1}\log n)}$. For sufficiently large n, we have

$$2^{O(n^{1.1}\log n)} \ll 2^{n^2},$$

so there must exist functions that cannot be computed by circuits of size $n^{1.1}$. Therefore, our algorithm will find such a function.

Solution to Problem 2

Claim. NEXP $\subseteq$ coNEXP$/_{n+1}$.

Let $L \in$ NEXP. We let $\alpha(n)$ be the number of yes-instances of L on inputs of length n. This can be specified in $n+1$ bits (since the range is from 0 to 2^n).

Let $x \notin L$. Given the correct advice $\alpha(n)$, we can verify that $x \notin L$ as follows:

- We guess a list of $\alpha(n)$ yes-instances of L on inputs of length n. For each of them we further guess their corresponding witnesses.
- We verify that all elements in this list are distinct, and that they are indeed yes-instances of L by verifying the witnesses. We reject immeidately if we find a duplicate or a witness that is not correct.
- We then check that our x is not in the list. If it is, we reject and accept otherwise.

We first note that given the correct advice $\alpha(n)$, the above procedure would proceed to step 3 if and only if the guessed list L is the the correct list of yes-instances of L on inputs of length n. Therefore, it would accept if and only if $x \notin L$. It also takes $2^{O(n)}$ non-deterministic time.

Therefore, we have that $L \in$ coNEXP$/_{n+1}$. Since L is arbitrary, we have that NEXP $\subseteq$ coNEXP$/_{n+1}$.

$\square$

Solution to Problem 3

Claim. CL $\subseteq$ ZPP.

Given an input x, we can think of the configuration of the machine as a pair (w, s), where $w \in \{0,1\}^{O(\log n)}$ is the clean workspace/pointer locations/state of the machine, and $s \in \{0,1\}^{\mathrm{poly}(n)}$ is the catalytic workspace/configuration of the machine.

Without loss of generality, we can assume the machine starts as $w_0 = 0^{O(\log n)}$ but an arbitrary $s_0 \in \{0,1\}^{\mathrm{poly}(n)}$. By the definition of CL, the machine should halt with the same starting s_0 in the catalytic workspace, but can have an arbitrary w in the clean workspace.

Let each pair (w, s) be a vertex, we can build a directed graph G where $(w, s) \to (w', s')$ if and only if the machine transition from configuration (w, s) to configuration (w', s') in one step.

Then, for each starting configuration (w_0, s_0), it corresponds to a path in G from (w_0, s_0) to some (w, s_0).

Here we make the key claim that for two different starting configurations (w_0, s_0) and (w_0', s_0'), the corresponding paths in G are disjoint.

Indeed, if they were not disjoint, then they would intersect at some configuration (w, s), and from thereafter the two paths would be the same. This contradicts the definition of CL since the machine should halt with the same starting s_0 in the catalytic workspace.

Therefore, all these $2^{|s|}$ paths must be disjoint. Since there are $2^{|w|} \cdot 2^{|s|}$ many states in total, the expected length of a random path is at most $2^{|w|} \cdot 2^{|s|}/2^{|s|} = 2^{|w|} \leq \mathrm{poly}(n)$.

Hence, our algorithm can work as follows:

1. Pick a random starting catalytic configuration s_0.
2. Simulate the machine starting from (w_0, s_0) until it halts and output the decision bit.

By the discussion above, the expected running time is at most $\mathrm{poly}(n)$. Therefore, we have that CL $\subseteq$ ZPP.

Solution to Problem 4

Claim. $\mathrm{TC}^1 \subseteq \mathrm{CL}$.

Proof. Let $L \in \mathrm{TC}^1$. Then for each input length n, there is a Boolean circuit C_n of depth $O(\log n)$ and polynomial size over threshold gates deciding L on inputs of length n. Each threshold gate G in C_n has fan-in $k = \mathrm{poly}(n)$ and computes

$$G(p_1, \ldots, p_k) = \mathbf{1}\left[\sum_{i=1}^{k} p_i = T\right]$$

for some threshold $T \in \{0, 1, \ldots, k\}$, where each $p_i \in \{0, 1\}$ is the output of a gate at the previous layer. (Note that although TC^1 is defined using majority gates, we can easily convert it to threshold gates by using the fact that

$$\mathrm{MAJ}(p_1, \ldots, p_k) = \sum_{T=\lceil k/2 \rceil}^{k} \mathbf{1}\left[\sum_{i=1}^{k} p_i = T\right].$$

Our goal is to simulate the evaluation of C_n by a clean register program that:

- uses only $O(\log n)$ *clean* bits (for the input, a program counter, and a recursion stack for the circuit of depth $O(\log n)$), and
- uses at most $\mathrm{poly}(n)$ *dirty* bits, which are all restored to their initial values at the end of the computation.

This will show that $L \in \mathrm{CL}$.

Registers and invariant. We work with registers $R_1, R_2, \ldots, R_M$ for some $M = \mathrm{poly}(n)$, each holding an integer modulo a prime p. We will fix p so that it is larger than the number of gates in C_n. All arithmetic is performed modulo p. For each gate G of C_n, we will construct a clean register program P_G such that, given a designated output register R_D,

- after running $P_G(D)$, all registers except R_D are restored to their original contents, and
- R_D is increased by the Boolean output of the gate G (viewed as 0 or 1).

We will construct $P_G(D)$ for each gate G in C_n in a bottom-up manner. First, for all input variables such programs can be constructed trivially. Now when we construct $P_G(D)$ for a gate G, we will make use of the programs $P_{G_1}(\cdot), \ldots, P_{G_k}(\cdot)$ for the input gates $G_1, \ldots, G_k$.

Step 1: Summation of fan-in values. Suppose G has fan-in k, with predecessor gates implemented by clean programs $P_1, \ldots, P_k$ (each P_i takes as argument the index of the register where it should add its output). Let $p_i \in \{0, 1\}$ be the output of the i-th predecessor gate.

We define an auxiliary program P_{sum} that, given a register index D, temporarily computes $\sum_{i=1}^{k} p_i$ and adds it into R_D, while restoring all other registers at the end:

$$P_{\mathrm{sum}}(D):$$

$$\begin{aligned}
\forall i \in \{1, \ldots, k\}: \quad & P_i(i) \\
& R_D := R_D + \sum_{i=1}^{k} R_i \\
\forall i \in \{1, \ldots, k\}: \quad & P_i^{-1}(i) \\
& R_D := R_D - \sum_{i=1}^{k} R_i
\end{aligned}$$

Let C_i denote the initial value of register R_i, and C_D the initial value of R_D. By correctness and cleanliness of each P_i, after the first loop we have $R_i = C_i + p_i$. Thus after the assignment $R_D := R_D + \sum_{i=1}^{k} R_i$,

$$R_D = C_D + \sum_{i=1}^{k}(C_i + p_i) = C_D + \sum_{i=1}^{k} C_i + \sum_{i=1}^{k} p_i.$$

After running P_i^{-1} for all i, each R_i is restored: $R_i = C_i$. Then

$$R_D := R_D - \sum_{i=1}^{k} R_i \quad \Rightarrow \quad R_D = C_D + \sum_{i=1}^{k} C_i + \sum_{i=1}^{k} p_i - \sum_{i=1}^{k} C_i = C_D + \sum_{i=1}^{k} p_i.$$

Thus all registers except R_D are restored, and R_D has been increased by $\sum_{i=1}^{k} p_i$.

Step 2: Shifting by the threshold. We next want to compute the difference $T - \sum_{i=1}^{k} p_i$. Let Q be an auxiliary register (dirty). We define a program P_{aux} that, given argument D and using Q, transforms R_D in such a way that the value $T - \sum_i p_i$ can be accessed cleanly while keeping other registers restored at the end:

$$P_{\text{aux}}(D) \text{ using register } Q:$$

$$\begin{aligned}
& P_{\text{sum}}(Q) \\
& R_D := R_D + T - R_Q \\
& P_{\text{sum}}^{-1}(Q) \\
& R_D := R_D + R_Q
\end{aligned}$$

Tracing values, write C_Q for the initial content of R_Q. After $P_{\text{sum}}(Q)$ we have

$$R_Q = C_Q + \sum_{i=1}^{k} p_i.$$

Thus after the update $R_D := R_D + T - R_Q$ we get

$$R_D = C_D + T - (C_Q + \sum_{i=1}^{k} p_i) = C_D + T - C_Q - \sum_{i=1}^{k} p_i.$$

Now running $P_{\text{sum}}^{-1}(Q)$ restores R_Q to C_Q, and leaves R_D unchanged. Finally, we do

$$R_D := R_D + R_Q$$

so that

$$R_D = C_D + T - C_Q - \sum_{i=1}^{k} p_i + C_Q = C_D + T - \sum_{i=1}^{k} p_i.$$

All other registers (including Q) are restored. Thus P_{aux} cleanly adds the quantity $T - \sum_i p_i$ into R_D.

Step 3: Computing a high power.

Let $\phi(p)$ be the Euler's totient function of p, note that $\phi(p) = p - 1$ if p is prime. Importantly, we have that (this uses the assumption on p that p is larger than the number of gates in C_n)

$$\left(T - \sum_i p_i\right)^{\phi(p)} = 0 \iff T - \sum_i p_i = 0,$$

and

$$\left(T - \sum_i p_i\right)^{\phi(p)} = 1 \iff T - \sum_i p_i \neq 0.$$

We would like to compute a high power $(T - \sum_i p_i)^{\phi(p)}$ for a suitable exponent $\phi(p)$, in a way that restores all registers except for a designated output register. Let us fix a prime p larger than an upper bound on T and k. We will compute $(T - \sum_i p_i)^{\phi(p)}$ using a standard "toggling" trick.

Let Q be as in the previous step, and introduce auxiliary registers $E, Q_1, \ldots, Q_{\phi(p)}$, all initially arbitrary (dirty). Define a program P_{EXP} that, given an input register D, uses these registers to add $(T - \sum_i p_i)^{\phi(p)}$ to R_E, while restoring all registers except E:

$$P_{\text{EXP}}(D) \text{ using registers } Q, E, Q_1, \ldots, Q_{\phi(p)}:$$

$P_{\text{aux}}^{-1}(Q)$

$\forall i \in \{1, \ldots, \phi(p)\}: \quad R_{Q_i} := R_{Q_i} + R_Q^i$

$P_{\text{aux}}(Q)$

$$R_E := R_E + \sum_{i=0}^{\phi(p)} \binom{\phi(p)}{i} (-1)^i R_{Q_i} R_Q^{\phi(p)-i}$$

$P_{\text{aux}}^{-1}(Q)$

$\forall i \in \{1, \ldots, \phi(p)\}: \quad R_{Q_i} := R_{Q_i} - R_Q^i$

$P_{\text{aux}}(Q)$

$$R_E := R_E - \sum_{i=0}^{\phi(p)} \binom{\phi(p)}{i} (-1)^i R_{Q_i} R_Q^{\phi(p)-i}$$

To see correctness, let $u := T - \sum_{i=1}^{k} p_i$, and let C_Q, C_{Q_i}, C_E be the initial contents of the corresponding registers. One checks (using that P_{aux} and P_{aux}^{-1} add and subtract u cleanly) that the net effect on Q and each Q_i is zero, and that the increment to R_E is

$$R_E \leftarrow C_E + \left[(C_Q - u)^{\phi(p)} - C_Q^{\phi(p)}\right].$$

Using the binomial theorem and cancellation of the terms depending only on C_Q, this simplifies (independently of the initial dirty contents) to

$$R_E = C_E + u^{\phi(p)} = C_E + (T - \sum_{i=1}^{k} p_i)^{\phi(p)}.$$

All other registers are restored to their initial values.

Step 4: Implementing a threshold gate. We now use P_{EXP} to implement the gate G.

Note that we actually need to compute $1 - (T - \sum_i p_i)^{\phi(p)}$ instead of $(T - \sum_i p_i)^{\phi(p)}$ in order to get the correct output of the threshold gate. This is because the output of the threshold gate is 1 if and only if $T - \sum_i p_i = 0$.

To realize this as a clean register program, let Q be as before and let D be the designated output register for G. Define $P_G(D)$ by:

$$P_G(D):$$

$$\begin{aligned}
&P_{\text{EXP}}(Q)\\
&R_D := R_D + 1 - R_Q\\
&P_{\text{EXP}}^{-1}(Q)\\
&R_D := R_D + R_Q
\end{aligned}$$

Exactly as in the previous toggling arguments, one checks that all registers other than D are restored to their original values, while R_D is increased by $1 - (T - \sum_i p_i)^{p-1} \in \{0, 1\}$, i.e., by the output of gate G.

Step 5: Simulating the whole circuit. Starting from the input bits (which we load into designated registers), we construct a clean program for each gate in the circuit layer by layer. For each gate we use a fresh output register (dirty) to accumulate its output, treating all other registers as potentially dirty but required to be restored by the end of the gate's program. Because:

- each gate program uses only $O(1)$ clean registers in addition to its input and output locations, and
- the circuit has depth $O(\log n)$,

the total clean space needed to simulate the recursion/stack of the circuit evaluation is $O(\log n)$. At any point, we are using only a polynomial number of registers (dirty), each of size $O(\log p) = O(\log n)$ bits, so the total dirty space is polynomial in n. By construction, at the end of the computation, all dirty registers are restored to their initial contents, and one designated output register holds the value of the circuit on the input x.

Thus we obtain a clean register program deciding L that uses $O(\log n)$ clean space and polynomially many dirty bits, i.e., $L \in \text{CL}$. Since $L \in \text{TC}^1$ was arbitrary, we conclude that $\text{TC}^1 \subseteq \text{CL}$. $\square$