1. ***1d-Clickomania.*** Consider the following game on a sequence of black-white tiles: In each step, you are allowed to choose any maximal run of *at least two* single-colored tiles, and remove them from the sequence. The rest of the tiles on the two sides will rejoin into a single sequence. If there is a way to remove all the tiles from the sequence, then we say that the sequence is *winnable*. For example,

$$\blacksquare\blacksquare\square\square\square\blacksquare\blacksquare\square \to \blacksquare\blacksquare\square\square\square\square \to \blacksquare\blacksquare \to \varepsilon$$

is winnable, while $\blacksquare\blacksquare\square\square\square\square\blacksquare\square$ is not.

Prove that the following language is not regular:

$$L = \left\{ w \in \{\blacksquare, \square\}^* : w \text{ is winnable} \right\}.$$

Solution: We prove that L is not regular by describing a fooling set for L of infinite size. Let

$$F := \left\{ (\blacksquare\square)^i : \text{for all integer } i \right\}.$$

Take two arbitrary elements in F, say without loss of generality $x := (\blacksquare\square)^i$ and $y := (\blacksquare\square)^j$ for $i < j$. Let $z := (\square\blacksquare)^i$.

- On one hand, we have $xz = (\blacksquare\square)^i(\square\blacksquare)^i = \varepsilon$ based on the rule. This shows that xz is winnable.
- On the other hand, we have $yz = (\blacksquare\square)^j(\square\blacksquare)^i$. As we attempt to simplify the tile sequence, at any moment there is only one monochromatic run with exactly two tiles. This implies the simplification process is unique and $yz = (\blacksquare\square)^{j-i}$, which is not winnable because there are no further tiles can be removed.

This implies that F is a fooling set for L of infinite size. ■

Rubric: Standard 5-point grading scaled to 10 points. 6 points if the fooling set is correct.

2. ***Fault-tolerant computing.*** Imagine you are given a good-old Turing machine with a single-tape and a single-head, but there is one catch: exactly one of the cells on the tape is *faulty*. Any symbol written on that cell might turn into other symbols unexpectedly, while the reading head is away. Worst of all, there are no ways to tell if the cell is faulty by examination, even after the cell has turned a symbol into another. (We assume that the faulty cell is not within the input.)

Prove that the faulty Turing machine can still simulate a standard Turing machine.

Solution: We describe how to emulate a standard Turing machine using a faulty Turing machine. The idea is to duplicate each symbol three times and read three consecutive cells at a time; now a majority vote always returns the correct symbol.

More precisely, we encode the symbol of the standard TM M by a duplicating the symbol three times in the faulty TM M'. ($\langle \texttt{0} \rangle = \texttt{000}$, $\langle \texttt{1} \rangle = \texttt{111}$, and $\langle \square \rangle = \square\square\square$). The input of M' is also encoded in the same fashion from the input of M. For each transition of M (that is, read a symbol, write a symbol, and move the head to an adjacent cell), the new machine M' read the corresponding three symbols, and treat the majority of the symbols as the symbol read. Then M' writes down three identical copies of the same symbol as required, and move the head three steps to the left or right.

To see the correctness of the emulation, because there is at most one faulty cell across the tape, the majority vote within the three cells will always return the correct symbol and thus M' correctly simulates the computation of M. ■

Rubric: Standard 5-point grading scaled to 10 points. Maximum 2 points if the majority vote idea is missing.

3. ***Inception.*** Let L_0 be an arbitrary regular language over alphabet Σ.

(a) Let D_0 be a DFA deciding L_0, given by some encoding $\langle D_0 \rangle$. Prove that the following language can be decidable by a Turing machine in polynomial time:

$$L_1 := \{ \langle D \rangle : L(D) = L(D_0) \},$$

where $L(D)$ denote the language recognized by DFA D.

Solution: To show that L_1 is in P, we prove that one can test whether the exclusive-or between $L(D)$ and $L(D_0)$ is empty on a given encoding of some DFA $\langle D \rangle$.

Testing if $\langle D \rangle$ encodes a valid DFA can be done in linear time. We construct DFA D' recognizing the language $L(D) \oplus L(D_0)$ — the exclusive-or between $L(D)$ and $L(D_0)$ — using product construction. Such DFA D' has size quadratic in D and D_0. To decide if $L(D')$ is empty, we can perform a graph-traversal from the starting state of D' and see if any accepting state can be reached. We accept $\langle D \rangle$ if the language $L(D')$ is empty.

Overall the running time is proportional to the size of D' which is polynomial in $|\langle D \rangle|$. ■

Rubric: Standard 5-point grading scaled to 10 points. Maximum 6 points if the construction of exclusive-or DFA is missing. Maximum 4 points if the algorithm is testing whether the *two DFAs* are equivalent.

(b) Let M_1 be a TM deciding L_1, given by some encoding $\langle M_1 \rangle$. Prove that the following language is undecidable:

$$L_2 := \{ \langle M \rangle : L(M) = L(M_1) \},$$

where $L(M)$ denote the language accepted by TM M.

Solution: We prove that L_2 is undecidable by reduction from the HALTING problem:

HALTING
- ***Input:*** *An encoding of a Turing machine $\langle M \rangle$, and a string w.*
- ***Output:*** *Does M run on input w halt?*

Assume for contradiction that L_2 can be decided by some TM M_2. We now construction another Turing machine H that decides HALTING, thus deriving a contradiction.

$H(\langle M \rangle, w)$:
construct Turing machine M_1' based on M and w
return $M_2(\langle M_1' \rangle)$

$M_1'(\langle D \rangle)$:
run M on w
return $M_1(\langle D \rangle)$

To prove that H decides the HALTING problem,

- If $(\langle M \rangle, w)$ is a yes-instance of HALTING, machine M_1' always terminates and mimics the behavior of M_1, which implies that $L(M_1') = L(M_1)$. Therefore M_2 on input M_1' will answer *yes*, implies that H accepts $(\langle M \rangle, w)$.
- If $(\langle M \rangle, w)$ is a no-instance of HALTING, machine M_1' runs forever, which implies that $L(M_1') = \varnothing \neq L(M_1)$ because $\langle D_0 \rangle$ must be in $L(M_1)$. Therefore M_2 on input M_1' will answer *no*, implies that H rejects $(\langle M \rangle, w)$.

Notice that we never really execute M_1' as a program but only as input to M_2, so H always terminates. ■

Rubric: Standard 5-point grading scaled to 10 points. Maximum 4 points if the reduction is incorrect or works for the wrong problem.

4. Consider the following problem:

> ZEROSUMSET
> - ***Input:*** *A set X of n integers in $[-N..N] \coloneqq \{-N, \ldots, 0, 1, \ldots, N\}$.*
> - ***Output:*** *Is there a nonempty subset S of X such that numbers in S sum up to 0?*

(a) Prove that ZEROSUMSET is NP-hard.

Solution: We prove that ZEROSUMSET is NP-hard by reduction from the standard NP-hard problem 3SAT.

Given a 3CNF-formula ϕ for 3SAT, we construct an instance X of ZEROSUMSET as follows. Let $x_1, \ldots, x_n$ be variables of ϕ and $C_1, \ldots, C_m$ be clauses of ϕ. We add integers of the form

$$\sum_{1 \le i \le n} \alpha_i \cdot 10^{i+m} + \sum_{1 \le j \le m} \beta_j \cdot 10^j$$

to X, represented as a vector $(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)$. Based on the following construction no sum of digits will ever overflow to other digits, and thus we can safely treat each base-10 integer as a vector.

- For each variable x_i, add integers a_i and $\overline{a_i}$ to X, corresponding to literals x_i and $\overline{x_i}$:
  - $\alpha_i = 1$ and $\alpha_{i'} = 0$ for every other i'.
  - $\beta_j = 1$ if clause C_j contains literal $x_i/\overline{x_i}$ and 0 otherwise, respectively.
- For each clause C_j, add two identical integers c_j whose only nonzero digit is at $\beta_j = -1$.
- Add the integer $-T = (-1, \ldots, -1, -3, \ldots, -3)$ to X.

This finishes the description of the (multi-)set X. To turn X into an actual set (without duplicate elements), it is sufficient to add different infinitesimals to each element and also include the negative of all the infinitesimals. The construction can be carried out in time polynomial to the size of the formula ϕ.

To prove that the reduction is correct:

- If ϕ is a *yes*-instance, we pick the numbers in X that correspond to the literals set to *true* in a fixed satisfying assignment of ϕ, together with $-T$ and enough c_js equal to one less than the number of true literals in each clause C_j, so that the total sum is zero. Exactly one literal per variable is picked, so each α_i digit sums to zero. Because the assignment is satisfying, at least one literal in each clause is true, and thus we have enough c_js so that the β_j digit sums up to 3, which is than cancelled by the -3 at the same digit in $-T$; thus each β_j also sums to zero.
- If ϕ is a *no*-instance, any assignment of ϕ has at least one clause with all literals being false. Any zero-sumset S of X we choose must contain $-T$ as it is the only number with negative values at digit α_is. Because each α_i sums to zero, exactly one a_i or $\overline{a_i}$ is chosen. Such choice give rise to an assignment of ϕ; let C_j denote the unsatisfied clause. Because $-T$ is chosen, the sum of digit β_j of the rest of $S \setminus \{-T\}$ has to be 3, which is impossible as no chosen numbers a_i have positive β_j and there are only two c_js.

This shows that an instance ϕ is satisfiable if and only if there is a non-empty subset of X summing up to zero. ■

Rubric: Standard 5-point grading scaled to 10 points. Maximum 4 points if instance construction does not work.

(b) Design an efficient algorithm that solves ZEROSUMSET, assuming we are given a subroutine that solves the SUMSET problem ("Given set Y and integer y, is there a nonempty subset summing up to y?") but only on *smaller instances*.

In other words, if the instance of ZEROSUMSET is a set X of n integers, then the subroutine can only be applied on set Y of size at most $n-1$.

Solution: Consider the following algorithm:

```
ZeroSumSet(X):
  For each element x in X:
      return yes if SumSet(X \ {x}, −x) = yes
  return no
```

The algorithm is correct because any non-empty zero-sumset S of X must contain at least one element x, and therefore the subset $S \setminus \{x\}$ must sum up to $-x$. Thus the oracle SumSet will correctly return *yes* when an element x in S is chosen during the for-loop. ■

Rubric: Standard 5-point grading.

(c) Design and analyze an algorithm that solves ZeroSumSet in $O(n^2N)$ time.

Solution: The idea is to implement the oracle queries by yourself. Because the input size is getting smaller and smaller, we can answer the query for the base case (when there is exactly one element in X) without trouble. If we record the results of all the past oracle queries, the recursion can be memoirized and is thus efficient.

We now describe the algorithm in details. Denote the elements in X as $x_1, \ldots, x_n$. As the absolute value of the sum of any subset of X cannot exceed nN, we will construct a Boolean-valued table SS with index $[n] \times [-nN \,..\, nN]$, where

$$SS[i][x] \coloneqq [\text{there is a nonempty subset of } \{x_1, \ldots, x_i\} \text{ that sums up to exactly } x].$$

All entries of SS are set to zero initially; we also assume the out-of-index values of SS are zero as well. Consider the following algorithm:

```
ZeroSumSet(X):
  for each i in [n]:
      for each x in [−nN .. nN]:
          SS[i][x] ← SS[i−1][x] or SS[i−1][x − x_i]
      SS[i][x_i] ← true
  return SS[n][0]
```

The algorithm correctly computes $SS[i][x]$ based on the recurrence relationship

$$SS[i][x] = \big(SS[i-1][x] \text{ or } SS[i-1][x-x_i]\big),$$

where the two terms correspond to choosing x_i in the subset or not.

There are n^2N entries in SS, and each takes $O(1)$ time to fill. So the total running time is $O(n^2N)$. ■

Rubric: Standard 5-point grading scaled to 10 points. Maximum 8 points if the algorithm allows empty set as solution. Maximum 4 points if the idea of storing past oracle query values / dynamic programming is missing.

(d) Imagine the following randomized algorithm: First we choose a random prime p of size $\sim n^4$, then fingerprint all numbers in X under modulo p and put them in Y. Now use the algorithm in (c) to solve the ZeroSumSet problem on Y in $O(n^5)$ time.

Why doesn't this solve ZeroSumSet in polynomial time with high probability?

Solution: Because N can be exponentially bigger than n, and a fingerprinting at order n^4 is not sufficient.

As an example, consider a *no*-instance X of size n that contains the element $P \coloneqq p_1 \cdot \cdots \cdot p_k$ where $\{p_1, \ldots, p_k\}$ are all the primes less than some Cn^4. There are about $\Theta(n^4/\log n)$ primes in this list and thus integer P has magnitude exponential in n. Any random prime p we choose for the finger printing will give rise to a set Y that always has a singleton set $\{P\}$ summing up to zero modulo p, and thus with probability 1 the algorithm fails. ■

Rubric: Standard 5-point grading. Maximum 3 points if the idea that N can be exponentially bigger than n is missing / no concrete example is given. Maximum 1 points if the idea that N can be way bigger than n is missing.