Homework7

April 6, 2026

1 Problem 1: Hamiltonian Cycle → SAT

In 1859, the Irish mathematician Sir William Rowan Hamilton invented a puzzle: find a path along the edges of a dodecahedron that visits every vertex exactly once and returns to the start. Today we know this as the Hamiltonian Cycle problem, one of Karp's original 21 NP-complete problems. It appears everywhere: a delivery truck visiting every stop and returning to the depot (the Traveling Salesman Problem), reconstructing a DNA sequence from overlapping fragments, and scheduling round-robin tournaments. Since the problem is NP-complete, no polynomial-time algorithm is known. But we do not need to solve NP-complete problems directly. We can ***reduce*** *them to SAT, and then unleash decades of engineering in modern SAT solvers (like Z3) to find solutions. In this problem, you will implement this reduction yourself.*

1.1 Background

Hamiltonian Cycle. Given an undirected graph $G = (V, E)$ with n vertices, a *Hamiltonian cycle* is a cycle that visits every vertex exactly once and returns to its starting vertex. More precisely, it is a sequence of distinct vertices $v_0, v_1, \ldots, v_{n-1}$ such that:

- Every vertex in V appears exactly once in the sequence
- $(v_i, v_{i+1}) \in E$ for all $i = 0, 1, \ldots, n-2$ (consecutive vertices are connected)
- $(v_{n-1}, v_0) \in E$ (the cycle wraps around: the last vertex connects back to the first)

NP-Completeness. Hamiltonian Cycle is NP-complete: it is in NP (given a candidate cycle, we can verify it in polynomial time) and every problem in NP can be reduced to it in polynomial time. This means there is no known polynomial-time algorithm for it, the best known exact algorithms run in exponential time.

Why reduce to SAT? Modern SAT solvers like Z3, MiniSat, and CaDiCaL can handle instances with millions of variables using sophisticated techniques (conflict-driven clause learning, unit propagation, restarts). By encoding Hamiltonian Cycle as a SAT instance, we can leverage all of this engineering instead of writing a backtracking search from scratch.

The position-based encoding. The key idea is to assign each vertex a *position* in the cycle. We create Boolean variables $x_{v,p}$ meaning "vertex v is at position p in the cycle." Then we add constraints ensuring that: 1. Each vertex gets exactly one position 2. Each position gets exactly one vertex 3. Consecutive positions in the cycle are connected by an edge

If the resulting formula is satisfiable, the satisfying assignment tells us the cycle. If it is unsatisfiable, no Hamiltonian cycle exists.

1.2 What You Must Implement

Submit a file named **`hw7_p1_hamcycle.py`** with the following function.

```
from z3 import *


def solve_hamiltonian_cycle(n: int, edges: list[tuple[int, int]]) -> dict:
    """
    Solve the Hamiltonian Cycle problem by reducing it to SAT using Z3.

    Given an undirected graph with n vertices (labeled 0 to n-1) and a list
    of edges, determine whether a Hamiltonian cycle exists. If it does,
    return the cycle as an ordered list of vertices.

    Encoding:
        - Bool variable x[v][p]: vertex v is at position p in the cycle
        - Exactly-one constraints per vertex and per position
        - Edge constraints: consecutive positions must be adjacent

    Parameters
    ----------
    n : int
        Number of vertices (labeled 0 to n-1).
    edges : list of tuple(int, int)
        Undirected edges of the graph.

    Returns
    -------
    dict with keys:
        'has_cycle' : bool: True if a Hamiltonian cycle exists
        'cycle'     : list[int] or None: vertices in cycle order (length n),
                      starting from vertex 0, or None if no cycle exists
    """
```

1.3 Hints

- **`exactly_one` helper.** Write a helper function that takes a list of Booleans and adds exactly-one constraints to the solver.
- **Adjacency set for $O(1)$ lookup.** Build a set of directed pairs for fast membership testing:

  ```
  adj = set()
  for u, v in edges:
      adj.add((u, v))
      adj.add((v, u))
  ```

- **Remember the wrap-around.** The cycle connects position $n-1$ back to position 0. Use `(p + 1) % n` for the next position index.
- **Extracting the cycle from the model.** Once Z3 returns `sat`, extract the cycle by iterating

over positions:

```
m = s.model()
cycle = [0] * n
for p in range(n):
    for v in range(n):
        if m.evaluate(x[v][p]):
            cycle[p] = v
```

- **Canonical starting vertex.** The autograder checks that your cycle is valid (all vertices present, all consecutive edges exist, wraps around). You may start from any vertex.

1.4 Sanity Checks

```
from hw7_p1_hamcycle import solve_hamiltonian_cycle

# Test 1: Triangle (0-1-2)
result = solve_hamiltonian_cycle(3, [(0,1), (1,2), (2,0)])
print(f"Triangle: {result}")
assert result['has_cycle'] == True
assert set(result['cycle']) == {0, 1, 2}
# Check edges: every consecutive pair (and wrap) should be an edge
edges_set = {(0,1),(1,0),(1,2),(2,1),(2,0),(0,2)}
cycle = result['cycle']
for i in range(len(cycle)):
    assert (cycle[i], cycle[(i+1) % len(cycle)]) in edges_set

# Test 2: Square (0-1-2-3)
result = solve_hamiltonian_cycle(4, [(0,1), (1,2), (2,3), (3,0)])
print(f"Square: {result}")
assert result['has_cycle'] == True

# Test 3: Star tree (no Hamiltonian cycle)
result = solve_hamiltonian_cycle(5, [(0,1), (0,2), (0,3), (0,4)])
print(f"Star tree: {result}")
assert result['has_cycle'] == False
assert result['cycle'] is None

# Test 4: Path graph (no Hamiltonian cycle)
result = solve_hamiltonian_cycle(5, [(0,1), (1,2), (2,3), (3,4)])
print(f"Path: {result}")
assert result['has_cycle'] == False

print("All sanity checks passed!")
```

2 Problem 2: Efficient Exactly-One Encoding

The exactly-one constraint is the bread and butter of SAT encodings. Every time you model a Sudoku puzzle ("each cell contains exactly one digit"), a scheduling problem ("each job is assigned exactly one time slot"), or a graph coloring ("each vertex gets exactly one color"), you are asserting that exactly one variable out of a group is true. The naive encoding works fine for small groups, but when a scheduling problem has 1,000 possible time slots, the quadratic blowup in clause count can overwhelm even modern SAT solvers. In Lectures 17 and 18, we learned a way to enforce it with $O(n^2)$ clauses. In this problem, you will implement both the naive and the efficient ladder (sequential) encoding, and see the difference.

2.1 Background

The **exactly-one constraint** over n Boolean variables $x_1, x_2, \ldots, x_n$ asserts that exactly one of them is true and all others are false. Formally:

$$\sum_{i=1}^{n} x_i = 1$$

This naturally decomposes into two parts:

- **At-least-one:** at least one variable is true: $x_1 \vee x_2 \vee \cdots \vee x_n$
- **At-most-one:** no two variables are simultaneously true

Both conditions together give exactly-one. The challenge is encoding at-most-one efficiently in CNF (conjunctive normal form), since SAT solvers operate on clauses.

2.2 Naive Pairwise Encoding

The simplest approach encodes at-most-one by forbidding every pair of variables from both being true.

At-least-one (1 clause):

$$x_1 \vee x_2 \vee \cdots \vee x_n$$

At-most-one ($\binom{n}{2} = \frac{n(n-1)}{2}$ clauses): for all $1 \leq i < j \leq n$:

$$\neg x_i \vee \neg x_j$$

Each such clause says "it is not the case that both x_i and x_j are true."

Total clause count:

$$C_{\text{naive}}(n) = 1 + \frac{n(n-1)}{2}$$

This is $O(n^2)$; manageable for small n, but for $n = 1{,}000$ the pairwise encoding alone requires 499,501 clauses. When a problem contains many exactly-one groups (as in Sudoku or scheduling), this quickly becomes prohibitive.

2.3 Ladder (Sequential) Encoding

The **ladder encoding** (also known as the sequential or Sinz encoding) achieves exactly-one using only $O(n)$ clauses by introducing auxiliary variables.

Auxiliary variables. Introduce n new Boolean variables $y_0, y_1, \ldots, y_{n-1}$ (0-indexed to match the implementation). The intended semantics of y_i is:

$$y_i = \text{true} \iff \text{at least one of } x_0, x_1, \ldots, x_i \text{ is true}$$

(Here we use 0-indexed $x_0, \ldots, x_{n-1}$ to match the Python implementation.)

Clauses. The encoding adds four groups of clauses:

1. **Implication from x to y** (for $i = 0, 1, \ldots, n-2$): "If x_i is true, then y_i must be true (a true value has been seen up to position i)." This contributes $n-1$ clauses.
2. **Propagation of y forward** (for $i = 0, 1, \ldots, n-2$): "If y_i is true, then y_{i+1} is true. If a true has been seen up to position i, it has also been seen up to position $i+1$." This contributes $n-1$ clauses.
3. **Exclusion: y_i blocks x_{i+1}** (for $i = 0, 1, \ldots, n-2$): "If a true has already been seen up to position i, then x_{i+1} must be false (at most one can be true)." This contributes $n-1$ clauses.
4. **At-least-one** (1 clause):
$$x_0 \vee x_1 \vee \cdots \vee x_{n-1}$$

Total clause count:

$$C_{\text{ladder}}(n) = 3(n-1) + 1 = 3n - 2$$

This is $O(n)$, a dramatic improvement over the naive encoding for large n.

2.4 What You Must Implement

Submit a file named **`hw7_p2_encoding.py`** with the following four functions.

```
from z3 import *

def exactly_one_naive(solver, variables) -> int:
    """
    Add the naive pairwise exactly-one encoding to the solver.

    Encoding:
        - At-least-one: Or(x_1, ..., x_n)
        - At-most-one: for each pair i < j, Or(Not(x_i), Not(x_j))
```

```
    Parameters
    ----------
    solver : z3.Solver
        The Z3 solver to add constraints to.
    variables : list of z3.BoolRef
        The boolean variables (at least 2).

    Returns
    -------
    int -- the number of clauses added to the solver.
    """


def exactly_one_ladder(solver, variables) -> int:
    """
    Add the ladder (sequential) exactly-one encoding to the solver.

    Encoding uses n auxiliary variables y_0, ..., y_{n-1} where y_i means
    "at least one of x_0, ..., x_i is true."

    Clauses (0-indexed, i = 0 to n-2):
        - Or(Not(x[i]), y[i])           ... x_i implies y_i
        - Or(Not(y[i]), y[i+1])         ... y_i implies y_{i+1}
        - Or(Not(y[i]), Not(x[i+1]))    ... y_i implies not x_{i+1}
        - Or(x[0], ..., x[n-1])         ... at-least-one

    Parameters
    ----------
    solver : z3.Solver
        The Z3 solver to add constraints to.
    variables : list of z3.BoolRef
        The boolean variables (at least 2).

    Returns
    -------
    int -- the number of clauses added to the solver.
    """
```

2.5 Hints

- **Naive at-most-one:** Use a double loop over all pairs `for i in range(n): for j in range(i+1, n):` and add `Or(Not(variables[i]), Not(variables[j]))` for each pair.
- **Ladder auxiliary variables:** Create n auxiliary `Bool` variables. A good naming convention is `f'_ladder_y_{i}'` to avoid name collisions. Index i runs from 0 to $n-2$ for all three clause groups.
- **The at-least-one clause is the same** in both encodings: `Or(variables)`.

2.6 Sanity Checks

```
from hw7_p2_encoding import exactly_one_naive, exactly_one_ladder
from z3 import *

# --- Correctness check (naive, n=3) ---
s = Solver()
xs = [Bool(f'x_{i}') for i in range(3)]
num = exactly_one_naive(s, xs)
assert num == 4, f"Expected 4 clauses, got {num}"

# Enumerate all solutions
solutions = []
while s.check() == sat:
    m = s.model()
    sol = [m.evaluate(x, model_completion=True) for x in xs]
    solutions.append(sol)
    s.add(Or([x != v for x, v in zip(xs, sol)]))

assert len(solutions) == 3, f"Expected 3 solutions for exactly-one on 3 vars, got {len(solutio
print("All sanity checks passed!")
```

3 Problem 3: Einstein's Zebra Puzzle

Einstein once said only 2% of people can solve this puzzle. Today, you join their ranks! The Zebra puzzle is a perfect showcase for constraint satisfaction. What makes the puzzle hard for humans: tracking dozens of interacting constraints across five categories, is exactly what SMT solvers excel at. In this problem, you will encode the classic Zebra puzzle as a constraint satisfaction problem using Z3's integer arithmetic and `Distinct` constraints, then let the solver find the unique solution in milliseconds.

3.1 Background

The Puzzle. Five houses stand in a row, numbered 1 through 5 from left to right. Each house has a unique nationality of its resident, a unique exterior color, a unique pet inside, a unique drink preferred by its resident, and a unique cigarette brand smoked by its resident. Given 15 clues, the goal is to determine the complete assignment, and in particular, answer two questions: *Who drinks water?* and *Who owns the zebra?*

Constraint Satisfaction. This puzzle is a classic *constraint satisfaction problem* (CSP). Each attribute must be assigned to exactly one house, every attribute within a category must go to a different house, and the 15 clues impose additional relational constraints. Rather than solving this by hand with tedious elimination, we can encode it directly in Z3.

Int-Variable Encoding. The key idea is elegant:

- Create one **integer variable** per attribute (e.g., Englishman, Red, Dog, etc.). The value of each variable represents the **house number** (1 through 5) where that attribute is assigned.
- For each category (nationality, color, pet, drink, cigarette), add a Distinct constraint ensuring no two attributes in the same category share a house.
- Each clue translates directly into a Z3 constraint.

If the solver returns `sat`, the model gives us the complete assignment. The Zebra puzzle has a unique solution.

3.2 The 15 Clues

The five categories and their attributes are:

Category	Attributes
Nationality	Englishman, Spaniard, Ukrainian, Norwegian, Japanese
Color	Red, Green, Ivory, Yellow, Blue
Drink	Coffee, Tea, Milk, Orange Juice, Water
Cigarette	Old Gold, Kools, Chesterfields, Lucky Strike, Parliaments
Pet	Dog, Snails, Fox, Horse, Zebra

The 15 clues are:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.

3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Old Gold smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house (house 3).
9. The Norwegian lives in the first house (house 1).
10. The man who smokes Chesterfields lives next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.
15. Someone owns a zebra.

The questions: Who drinks water? Who owns the zebra?

3.3 Encoding Strategy

Each attribute becomes a Z3 `Int` variable whose value is the house number (1–5):

$$\text{Englishman} \in \{1,2,3,4,5\}, \quad \text{Red} \in \{1,2,3,4,5\}, \quad \ldots$$

Range constraints. For every variable v:

$$1 \leq v \leq 5$$

Distinctness. For each category, all variables must take different values. For example, for nationalities:

$$\text{Distinct}(\text{Englishman}, \text{Spaniard}, \text{Ukrainian}, \text{Norwegian}, \text{Japanese})$$

Since there are 5 variables each constrained to $\{1, \ldots, 5\}$ and all distinct, this is equivalent to saying each variable gets a unique house, a perfect assignment.

3.4 What You Must Implement

Submit a file named **`hw7_p3_zebra.py`** with the following function.

```
from z3 import *

def solve_zebra() -> dict:
    """
    Solve Einstein's Zebra Puzzle using Z3 SMT solver.

    Five houses in a row, each with a unique nationality, house color, pet,
    drink, and cigarette brand. Given 15 clues, determine: who drinks water
    and who owns the zebra?
```

```
    Encoding:
        - Each attribute (e.g., 'Englishman', 'Red', 'Dog') is an Int variable
          representing the house number (1-5).
        - Distinct constraint per category.
        - Each clue becomes a Z3 constraint.

    Returns
    -------
    dict with keys:
        'water_drinker' : str -- nationality of the water drinker
        'zebra_owner'   : str -- nationality of the zebra owner
        'assignment'    : dict mapping attribute name (str) to house number (int)
    """
```

The `assignment` dictionary must map **every attribute name** (a string) to its **house number** (an integer, 1–5). The expected attribute name strings are:

- Nationalities: `'Englishman'`, `'Spaniard'`, `'Ukrainian'`, `'Norwegian'`, `'Japanese'`
- Colors: `'Red'`, `'Green'`, `'Ivory'`, `'Yellow'`, `'Blue'`
- Drinks: `'Coffee'`, `'Tea'`, `'Milk'`, `'OrangeJuice'`, `'Water'`
- Cigarettes: `'OldGold'`, `'Kools'`, `'Chesterfields'`, `'LuckyStrike'`, `'Parliaments'`
- Pets: `'Dog'`, `'Snails'`, `'Fox'`, `'Horse'`, `'Zebra'`

3.5 Sanity Checks

```
from hw7_p3_zebra import solve_zebra

result = solve_zebra()
a = result['assignment']

# Check 1: The two answers
print(f"Water drinker: {result['water_drinker']}")
print(f"Zebra owner:   {result['zebra_owner']}")
assert result['water_drinker'] == 'Norwegian', f"Expected Norwegian, got {result['water_drinke
assert result['zebra_owner'] == 'Japanese', f"Expected Japanese, got {result['zebra_owner']}"

# Check 2: 25 attributes total, all in range [1, 5]
assert len(a) == 25, f"Expected 25 attributes, got {len(a)}"
for name, house in a.items():
    assert 1 <= house <= 5, f"{name} assigned to house {house}, not in [1, 5]"

# Check 3: Distinct per category
nationalities = ['Englishman', 'Spaniard', 'Ukrainian', 'Norwegian', 'Japanese']
assert len(set(a[n] for n in nationalities)) == 5, "Nationality houses not distinct"

# Check 4: A few specific clues
assert a['Englishman'] == a['Red'],   "Clue 1 violated"
assert a['Spaniard']   == a['Dog'],   "Clue 2 violated"
assert a['Norwegian']  == 1,          "Clue 9 violated"
```

```
assert a['Milk']       == 3,          "Clue 8 violated"
assert a['Green'] == a['Ivory'] + 1,  "Clue 5 violated"

print("All sanity checks passed!")
```

4 Problem 4: Verifying Bit-Manipulation Algorithms

Modern systems code is full of clever bit-manipulation tricks. The HAKMEM memo from MIT (1972) contains dozens of them; Google's Abseil library, the Linux kernel, and hash-table implementations all rely on non-obvious bitwise identities for performance. But how do we know these tricks are correct? Testing helps, but 32-bit integers have over four billion possible values so exhaustive testing is slow, and random testing can miss edge cases. Here is the key insight from Lectures 19-20: we can use an SMT solver like Z3 to ***verify*** *bit-manipulation algorithms over all possible inputs simultaneously. By modeling unsigned 32-bit arithmetic exactly with* `BitVec(32)`*, we can prove equivalence, find counterexamples, and check algebraic properties, all automatically. In this problem you will use Z3's bit-vector reasoning to verify (or refute) three real-world bit tricks.*

4.1 Background

Bit tricks in systems code. Programmers frequently use bitwise operations to implement arithmetic more efficiently. Classic examples include: - Computing the average of two integers without overflow - Rounding up to the next power of two (used in hash-table resizing) - Mixing hash values to improve distribution (used in Murmur3, xxHash, etc.)

These tricks are fast, but their correctness is often non-obvious. A single wrong shift amount or an off-by-one constant can introduce subtle bugs that only manifest for specific inputs.

Z3 BitVec(32) models unsigned 32-bit arithmetic exactly. Z3's `BitVec` sort represents fixed-width bit vectors. A `BitVec("x", 32)` variable models a 32-bit unsigned integer. All arithmetic operations (`+`, `-`, `*`, `&`, `|`, `^`) are performed modulo 2^{32}, exactly matching hardware behavior. This means we can translate Python bit-manipulation code directly into Z3 expressions and reason about all 2^{32} possible inputs at once.

4.2 The Negation Trick

The fundamental technique for verification with SMT solvers is the **negation trick**:

> **To prove that property P holds for all inputs, assert $\neg P$ and check satisfiability.** - If the solver returns **UNSAT**: $\neg P$ has no solution, so P holds for all inputs. **Property proved.** - If the solver returns **SAT**: the model is a **counterexample**, a concrete input where P fails.

Example: Proving equivalence. To prove $f(x) = g(x)$ for all 32-bit x: 1. Create a Z3 `BitVec("x", 32)`. 2. Compute $f(x)$ and $g(x)$ as Z3 expressions. 3. Assert $f(x) \neq g(x)$. 4. If UNSAT: $f \equiv g$ for all inputs. If SAT: the model gives a counterexample.

4.3 Z3 BitVec Cheat Sheet

When translating Python bit-manipulation code to Z3 `BitVec` expressions, the following mappings apply:

Python Operation	Z3 BitVec Equivalent	Notes
`a >> k`	`LShR(a, k)`	**Logical** shift right (fills with 0s). Do NOT use `>>` which is arithmetic (sign-extending).
`a & b`	`a & b`	Z3 overloads `&` for bit-vectors.
`a ^ b`	`a ^ b`	Z3 overloads `^` for bit-vectors.
`a | b`	`a | b`	Z3 overloads `|` for bit-vectors.
`a * b`	`a * b`	Z3 auto-truncates the result to the bit-vector width.
`a + b`	`a + b`	Wraps modulo 2^{32} (unsigned overflow).
`a - b`	`a - b`	Wraps modulo 2^{32} (unsigned underflow).
`a < b` (unsigned)	`ULT(a, b)`	**Unsigned** less-than. Do NOT use `<` (signed).
`a <= b` (unsigned)	`ULE(a, b)`	**Unsigned** less-than-or-equal. Do NOT use `<=` (signed).
`a > b` (unsigned)	`UGT(a, b)`	**Unsigned** greater-than.
`a >= b` (unsigned)	`UGE(a, b)`	**Unsigned** greater-than-or-equal.

4.4 Task A: Average Without Overflow (30 pts)

Consider two implementations of unsigned integer average:

v1:

```
def avg_v1(a, b):
    return (a & b) + ((a ^ b) >> 1)
```

v2:

```
def avg_v2(a, b):
    return (a + b) >> 1
```

Your tasks: 1. Prove v1 and v2 are either equivalent or NOT equivalent using Z3. If they are not equivalent, return a concrete counterexample (a, b). 2. Provide a Z3 proof that v1 always returns a value between $\min(a, b)$ and $\max(a, b)$ (inclusive).

4.5 Task B: Next Power of Two (30 pts)

The "next power of two" operation rounds an unsigned integer up to the nearest power of 2 (or returns the input if it is already a power of 2). This is commonly used when resizing hash tables.

v1:

```
def npo2_v1(x):
    x = x - 1
    x = x | (x >> 1)
    x = x | (x >> 2)
```

```
    x = x | (x >> 4)
    x = x | (x >> 8)
    x = x | (x >> 16)
    return x + 1
```

v2:

```
def npo2_v2(x):
    x = x | (x >> 1)
    x = x | (x >> 2)
    x = x | (x >> 4)
    x = x | (x >> 8)
    x = x | (x >> 16)
    return x + 1
```

Your tasks: 1. Prove v1 and v2 are NOT equivalent. Return a counterexample. 2. Which implementation is wrong, and which implementation is correct?

Hint: Checking "is a power of 2." A positive integer n is a power of 2 if and only if $n \& (n-1) = 0$. In Z3: `result & (result - 1) == 0`.

4.6 Task C: Hash Mix Verification (40 pts)

Hash mixing functions are used to improve the distribution of hash values. The **Murmur3 fmix32** function is a widely-used finalizer:

v1 (standard Murmur3 fmix32):

```
def mix_v1(x):
    x ^= x >> 16
    x = (x * 0x85ebca6b) & 0xFFFFFFFF
    x ^= x >> 13
    x = (x * 0xc2b2ae35) & 0xFFFFFFFF
    x ^= x >> 16
    return x
```

v2 (buggy – one constant changed by +1):

```
def mix_v2(x):
    x ^= x >> 16
    x = (x * 0x85ebca6b) & 0xFFFFFFFF
    x ^= x >> 13
    x = (x * 0xc2b2ae36) & 0xFFFFFFFF   # Off by one!
    x ^= x >> 16
    return x
```

The constants in Murmur3 were carefully chosen to make fmix32 a **bijection** (one-to-one and onto). Changing even a single bit in a constant can destroy this property.

Your tasks: 1. Prove v1 and v2 are NOT equivalent. Return a counterexample. 2. Verify that v1 has a **fixed point**: a value x such that mix_v1$(x) = x$. 3. (Hard) Verify that v1 is a **bijection**: if mix_v1$(a) =$ mix_v1(b), then $a = b$ but that v2 is not.

4.7 What You Must Submit

Submit a pdf of your answers to all subparts.

4.8 Hints

- **For "between min and max":** Use `ULE` and `UGE` for unsigned comparison. The property to verify is:

  `ULE(min_ab, avg) and ULE(avg, max_ab)`

  where `min_ab` and `max_ab` can be expressed using `If(ULE(a, b), a, b)` and `If(ULE(a, b), b, a)`.

- **For fixed point:** Assert `mix_v1(x) == x` and check for SAT. If SAT, extract x from the model.

5 Problem 5: Neuroevolution for LunarLander

In Lecture 16 we saw how Genetic Algorithms optimize numeric vectors via selection, mutation, and crossover. The key insight was that these operators work on any "genome", a vector of numbers that encodes a candidate solution. We applied them to continuous optimization and to the TSP. But what if the genome was the weight vector of a neural network? In this problem, you will implement a Genetic Algorithm that evolves a neural network policy to land a simulated spacecraft.

5.1 What is a Neural Network?

If you have never seen a neural network before, do not worry. For the purposes of this problem, a neural network is just a **parameterized function**: a function whose behavior is determined by a collection of numbers (called *weights* and *biases*) that we get to choose.

The architecture we use. Our network takes an input vector $x \in \mathbb{R}^8$ (the state of the lunar lander) and produces an output vector $y \in \mathbb{R}^4$ (a score for each possible action of the lunar landing). The computation is:

$$y = W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2$$

where:

- $W_1 \in \mathbb{R}^{64\times 8}$ is the first weight matrix (64 rows, 8 columns)
- $b_1 \in \mathbb{R}^{64}$ is the first bias vector
- $\text{ReLU}(z) = \max(0, z)$ is applied element-wise (it simply replaces negative values with zero)
- $W_2 \in \mathbb{R}^{4\times 64}$ is the second weight matrix
- $b_2 \in \mathbb{R}^4$ is the second bias vector

How many parameters? Count them up:

Parameter	Shape	Count
W_1	64×8	512
b_1	64	64
W_2	4×64	256
b_2	4	4
Total		**836**

So the network is fully determined by 836 numbers. If we flatten all of these into a single vector $\theta \in \mathbb{R}^{836}$, then choosing θ determines the function $f_\theta : \mathbb{R}^8 \to \mathbb{R}^4$.

> **Key insight:** A neural network is just a parameterized function. *Learning* means finding good parameter values. Normally this is done via gradient descent (backpropagation), but we can also do it with a Genetic Algorithm, just treat θ as the genome.

No backpropagation needed! In a typical machine learning course, you would learn to compute gradients of a loss function with respect to the weights and use gradient descent to update them. We skip all of that. Instead, the GA treats the 836-dimensional weight vector as a genome and optimizes it using the same operators from Lecture 16: tournament selection, blend crossover, and

Gaussian mutation. The fitness function is simply "how well does this network play LunarLander?"

5.2 What is Reinforcement Learning?

Reinforcement learning (RL) is a framework for sequential decision-making:

1. An **agent** observes the current **state** of the environment.
2. The agent picks an **action**.
3. The environment transitions to a new state and returns a **reward**.
4. Repeat until the episode ends.

The agent's goal is to maximize the **total reward** accumulated over an episode.

Policy. A *policy* is a function that maps states to actions: $\pi : \text{State} \to \text{Action}$. Our neural network IS the policy: given a state vector (8 floats) representing the state of the Lunar Lander, it outputs scores for 4 actions, and we pick the action with the highest score.

$$\pi(s) = \arg\max_a \ f_\theta(s)[a]$$

Fitness = total episode reward. In a standard GA, each individual has a fitness value. Here, the fitness of a neural network (with weights θ) is the total reward it accumulates when playing an episode of LunarLander from start to finish:

$$\text{fitness}(\theta) = \sum_{t=0}^{T} r_t$$

where r_t is the reward at time step t and T is the number of steps until the episode ends.

> **Think of it this way:** the GA's fitness function = total episode reward. Tournament selection, crossover, and mutation work on the weight vector exactly as in Lecture 16 for continuous optimization. The only difference is that evaluating fitness requires running a simulation instead of computing a formula.

5.3 LunarLander Environment

We use the classic **LunarLander-v3** environment from the Gymnasium library. A spacecraft starts at the top of the screen and must land safely on a flat landing pad.

State (8 floats). At each time step, the agent observes:

Index	Observation	Range (approx.)
0	x position	$[-1.5, 1.5]$
1	y position	$[0, 1.5]$
2	x velocity	$[-5, 5]$
3	y velocity	$[-5, 5]$
4	angle	$[-\pi, \pi]$
5	angular velocity	$[-5, 5]$
6	left leg contact	$\{0, 1\}$
7	right leg contact	$\{0, 1\}$

Actions (4 discrete). At each time step, the agent chooses one action:

Action	Meaning
0	Do nothing
1	Fire left orientation engine
2	Fire main engine
3	Fire right orientation engine

Reward structure:

- **Landing on the pad:** +100 to +140 points (closer to center = more reward)
- **Crashing:** -100 points
- **Each leg contact with ground:** +10 points
- **Firing main engine:** -0.3 per frame
- **Firing side engine:** -0.03 per frame
- **Moving away from pad:** small negative reward proportional to distance

Episode termination:

- The lander lands (both legs contact ground and velocity is low) — **success**
- The lander crashes (body contacts ground too hard) — **failure**
- 1000 time steps elapse — **truncated**

What does "solved" mean? An average reward of +200 over 100 consecutive episodes is considered "solved." For this assignment, we use graduated thresholds (see grading below).

5.4 Neuroevolution: GA + Neural Networks

Neuroevolution applies the Genetic Algorithm from Lecture 16 to neural network weight vectors. Here is the mapping between GA concepts and our neural network setting:

GA Concept	Neuroevolution Equivalent
Individual	A neural network with specific weights
Genome	The 836-float weight vector θ
Population	A list of 50 neural networks
Fitness	Average reward over N episodes
Selection	Tournament selection (same as Lecture 16)
Crossover	Blend crossover on weight tensors (same as Lecture 16)
Mutation	Add Gaussian noise to weights (same as Lecture 16)
Elitism	Keep the top 2 unchanged

Recall from Lecture 16:

- **Tournament selection:** pick k individuals at random from the population, return the one with the highest fitness. This provides selection pressure (better individuals are more likely to be chosen) while still giving weaker individuals a chance.

- **Blend crossover:** given parents $p_1, p_2 \in \mathbb{R}^n$, produce:

$$\text{child} = \beta \cdot p_1 + (1 - \beta) \cdot p_2, \quad \beta \sim \text{Uniform}(0, 1)$$

- **Gaussian mutation:** add noise to the child:

$$\text{child}_{\text{mutated}} = \text{child} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I)$$

This is EXACTLY the GA from Lecture 16.** The only difference: instead of optimizing a Rastrigin's function, you are optimizing an 836-dimensional vector for LunarLander reward. The operators are identical.

Training loop pseudocode:

```
Initialize population of N random neural networks (N different thetas)
For each generation:
    1. Evaluate fitness of every individual (average reward over 3 episodes)
    2. Record the best-ever individual
    3. Create next generation:
       a. Copy top ELITISM individuals unchanged
       b. Fill remaining slots:
          - Select parent1 via tournament selection
          - Select parent2 via tournament selection
          - child = crossover(parent1, parent2)
          - mutate(child)
          - Add child to next generation
    4. Replace population with next generation
Save best-ever model weights
```

5.5 What You Must Implement

File: `hw7_p5_lander.py`

You must implement three functions. All three operate on `LunarPolicy` objects (PyTorch `nn.Module` instances). The network architecture is fixed and provided: you only need to implement the GA operators.

5.5.1 Function 1: `tournament_select(population, fitnesses, k=5)`

Select the best individual from k randomly chosen members of the population.

Algorithm: 1. Sample k indices uniformly at random (without replacement) from $\{0, 1, \ldots, |\text{population}| - 1\}$. 2. Among these k individuals, find the one with the highest fitness. 3. Return a **deep copy** of that individual (so mutations on the child do not affect the original).

Parameters: - `population`: list of `LunarPolicy` objects - `fitnesses`: list of floats (same length as `population`) - `k`: int, tournament size (default 5)

Returns: `LunarPolicy`, a deep copy of the tournament winner.

5.5.2 Function 2: `mutate(model, sigma=0.1)`

Add Gaussian noise to every parameter (weight and bias) in the model.

For each parameter tensor p in the model:

$$p \leftarrow p + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2) \text{ (element-wise)}$$

This is the same Gaussian mutation from Lecture 16. Use `torch.randn_like(p) * sigma` to generate noise with the right shape.

Parameters: - `model`: `LunarPolicy` (modified **in-place**) - `sigma`: float, standard deviation of noise (default 0.1)

Returns: `LunarPolicy`, the same model object (modified in-place).

Hint: Loop over `model.parameters()` and use `p.data.add_(...)` to modify in-place.

5.5.3 Function 3: `crossover(parent1, parent2)`

Create a child by blending parent weights using blend crossover.

Algorithm: 1. Create a new `LunarPolicy` child. 2. For each parameter tensor (there are 4: W_1, b_1, W_2, b_2): - Sample $\beta \sim \text{Uniform}(0, 1)$ (one scalar per parameter tensor) - Set child parameter: $\text{child}_p = \beta \cdot \text{parent1}_p + (1 - \beta) \cdot \text{parent2}_p$

Parameters: - `parent1`: `LunarPolicy` - `parent2`: `LunarPolicy`

Returns: `LunarPolicy`, a new child model.

Hint: Use `zip(child.parameters(), parent1.parameters(), parent2.parameters())` to iterate over corresponding parameter tensors. Use `torch.rand(1).item()` to sample β.

5.6 What We Provide

The following are already implemented in `hw7_p5_lander.py`:

- **`LunarPolicy` class:** The fixed neural network architecture ($8 \to 64 \to 4$). Do not modify this.
- **`evaluate_policy(model, env, n_episodes=3):`** Runs the model in the environment for `n_episodes` episodes and returns the average total reward. This is the fitness function.
- **Training loop:** The main training loop that initializes a population, evolves it for multiple generations, and saves the best model.

Additionally, we provide:

- **`lunar_lander_env.py`:** A wrapper around the Gymnasium LunarLander-v3 environment.

Dependencies. Before running, install the required packages:

```
pip install numpy torch gymnasium[box2d]
```

On some systems, you may also need `swig` installed (it is a build dependency for Box2D):

```
# macOS
brew install swig
```

```
# Ubuntu/Debian
sudo apt-get install swig
```

5.7 Suggested Hyperparameters

The following hyperparameters work well as a starting point. You are free to modify them in the training loop.

Hyperparameter	Suggested Value	Notes
Population size	50	Larger = more diversity, but slower
Generations	100	Increase if fitness is still improving
Mutation σ	0.1	Too large = chaotic; too small = slow
Tournament k	5	Larger = more selection pressure
Elitism	2	Always keep the top 2 unchanged
Eval episodes	3	Average over 3 to reduce noise

Expected training time: 5–15 minutes on a modern laptop (depending on population size and number of generations). The bottleneck is running LunarLander simulations for fitness evaluation, each episode runs up to 1000 physics steps.

Expected performance: With the suggested hyperparameters, you should be able to achieve an average reward above 150 within 100 generations. If your lander consistently lands on the pad, you may see rewards above 200.

5.8 Hints and Tips

1. **Average over multiple episodes for fitness.** The LunarLander environment is stochastic (the initial state and wind vary between episodes). Evaluating on a single episode is very noisy, a lucky starting position might make a bad policy look good, or vice versa. Average over at least 3 episodes.

2. **If no improvement after 20 generations,** try:

   - Increasing population size (e.g., 100)
   - Increasing mutation σ (e.g., 0.2)
   - Increasing tournament size k (more selection pressure)
   - Running for more generations

3. **Save the best model each generation.** The training loop already does this via `best_ever_model`. The best-ever individual across all generations is what gets saved, so even if a later generation is worse, you keep the all-time best.

4. **This is the same GA from Lecture 16.** Recall the slide on continuous optimization: blend crossover produces child $= \beta \cdot p_1 + (1 - \beta) \cdot p_2$ with $\beta \sim U(0, 1)$, and mutation adds $\mathcal{N}(0, \sigma^2)$ noise.

5. **Watch the training output.** You should see the best fitness and average fitness generally increase over generations. A typical successful run might look like:

```
Gen   0 | best    -83.7 | avg   -494.3 | all-time best    -83.7
Gen  10 | best    -57.2 | avg   -193.7 | all-time best    -18.0
Gen  30 | best     27.2 | avg   -144.8 | all-time best     27.2
Gen  60 | best     99.1 | avg   -122.3 | all-time best    120.5
Gen  99 | best     71.8 | avg   -104.9 | all-time best    158.6
```

6. **You can visualize the trained policy** by creating an environment with `render_mode="human"`:

```
from lunar_lander_env import LunarLanderEnv
env = LunarLanderEnv(render_mode="human")
# load and run your model...
```

5.9 Sanity Checks

Before submitting, verify:

- ☐ Your three functions (`tournament_select`, `mutate`, `crossover`) do not raise exceptions.
- ☐ `tournament_select` returns a deep copy (modifying the returned model does not change the population).
- ☐ `mutate` modifies weights in-place and returns the model.
- ☐ `crossover` returns a new `LunarPolicy` with blended weights.
- ☐ The saved `lunar_policy.pt` file loads correctly:

```
model = LunarPolicy()
model.load_state_dict(torch.load("lunar_policy.pt", map_location="cpu", weights_only=True)
```

- ☐ The model has exactly 836 parameters.
- ☐ Run `evaluate_policy` on your saved model, average reward should be above 50 at minimum (above 150 for full credit).

5.10 Grading and what to submit:

Just submit `lunar_policy.pt` after training. The autograder will load your submitted `lunar_policy.pt`, evaluate it over 20 episodes with fixed random seeds, and assign points based on the average reward:

Test	Criterion	Points
5.1	Model loads with correct architecture (836 params)	10
5.2	Average reward > -200 (the lander flies)	15
5.3	Average reward > -50 (it tries to land)	25
5.4	Average reward > 50 (it usually lands)	25
5.5	Average reward > 150 (it lands well)	25
Total		**100**

Points are cumulative, a model scoring above 150 earns all 100 points.

5.11 Submission

Submit the following file to Gradescope:

- `lunar_policy.pt` — your trained model weights

You do NOT need to submit `hw7_p5_lander.py` (we only grade the model performance). However, make sure your model was trained using the `LunarPolicy` architecture provided, as the autograder uses the same architecture to load the weights.