Directions:

- Submit written answers via Gradescope.
- Submit the SMT file for Problem 3 via the separate Gradescope assignment.
- Complete the corresponding Gradiance quizzes by the due date. Note that the “pts” and “hpts” relations in Gradiance questions are equivalent to “vP” and “hP” discussed in class.
- You may use up to two of your remaining late days for this assignment, for a late deadline of March 17, 2023 at 11:59pm. **There are no late days for Gradiance.**
- This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.
Problem 1. Pointer Analysis.

Consider the following Java snippet where each object allocation is labeled. Answer the following questions.

```java
public class A {
    A f = null;

    public A foo() {
        return new A(); // h1
    }

    public static A bar(A p, A q) {
        if (p != q) {
            p = q;
        } else {
            p = new B(); // h2
            p.f = q;
        }
        return p;
    }
}

public class B extends A {
    public A foo() {
        return new B(); // h3
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A(); // h4
        A b = a.foo();
        a = A.bar(a, b);
    }
}
```

1. What is the perfect set of allocations that variable `a` can refer to at any point while actually executing this program?

   `{h1, h4}`. `h4` comes from the first assignment, and `h1` comes from lines 10 and 15 in `bar` where `p = a` and `q = b`, and `b` was allocated from `h1` (lines 28 and 5).
2. What are the vP and hP tuples inferred from this code in a context-insensitive, flow-insensitive pointer analysis?

Two approaches. The first determines the call graph on the fly (please refer to slide number 40 of lecture 12):

- vP(a, h4): From line 27 (initial assignment).
- vP(b, h1): We construct the call graph on the fly and currently a is allocated at h4, so a has type A, and A’s foo should be called
- vP(p, h4): From line 29; set to all things a can be.
- vP(q, h1): From line 29; set to all things b can be.
- vP(p, h1): From line 10; set to all things q can be (flow-insensitive).
- vP(p, h2): From line 12 (flow-insensitive).
- vP(a, h1):
- vP(a, h2): From lines 15 and 29; set to all things p can be (h4 is already mentioned above).

Note that then the call graph is updated because a could also be allocated at h2 so it could have type B, then at line 28, B’s foo could be called.

We add:
- vP(b, h3)
and propagate it to:
- vP(q, h3)
- vP(a, h3)
- vP(p, h3)

Finally derive the hP tuples:

- hP(h1, f, h1):
- hP(h1, f, h3):
- hP(h2, f, h1):
- hP(h2, f, h3):
- hP(h3, f, h1):
- hP(h3, f, h3):
- hP(h4, f, h1):

- hP(h4, f, h3): From line 13; first argument set to all things p can be, third argument set to all things q can be.

The second uses a fixed call graph (but yields the same result):

- vP(a, h4): From line 27 (initial assignment).
- vP(b, h1):
- vP(b, h3): From line 28; we only know a is of type A, but not the dynamic type (could be A or B). Note: you don’t to reason about the hType or invokes tuples.
- vP(p, h4): From line 29; set to all things a can be.
- vP(q, h1):
- vP(q, h3): From line 29; set to all things b can be.
- vP(p, h1):
vP(p, h3): From line 10; set to all things q can be (flow-insensitive).
vP(p, h2): From line 12 (flow-insensitive).
vP(a, h1):
vP(a, h2):
vP(a, h3): From lines 15 and 29; set to all things p can be (h4 is already mentioned above).

hP(h1, f, h1):
hP(h1, f, h3):
hP(h2, f, h1):
hP(h2, f, h3):
hP(h3, f, h1):
hP(h3, f, h3):
hP(h4, f, h1):
hP(h4, f, h3): From line 13; first argument set to all things p can be, third argument set to all things q can be.
**Problem 2. Binary Decision Diagram.**

In this problem, we will draw the BDD for the following expression in a few steps:

\[ B = \forall x_0: (x_0 \land x_2) \oplus (\neg x_1 \lor x_3). \]

Here, \( \oplus \) refers to the XOR logical operation.

1. Draw the BDD for \( B_0 = (x_0 \land x_2) \oplus (\neg x_1 \lor x_3) \) with the following variable order: \( x_0, x_2, x_1, x_3 \).

   The following is the ROBDD under order: \( x_0, x_2, x_1, x_3 \). Drawing it as a complete binary tree and not simplifying it at all is acceptable.

   ![Diagram](image)

2. Draw the BDD for \( B \) using the diagram for \( B_0 \). Do not simplify the resulting BDD – you will do so in the next step.

   Notice that \( \forall x: P(x) \) is equivalent to \( P(0) \land P(1) \). Thus, we have:

   Restricting \( B_0 \) to \( x_0 = 0 \):

   ![Diagram](image)

   Restricting \( B_0 \) to \( x_0 = 1 \):

   ![Diagram](image)

   Applying AND:
Note that if you drew a complete binary tree in part 1, the answer for part 2 should also be a complete binary tree (height = 3 instead of 4 though since we got rid of $x_0$).

3. Collapse redundant nodes in your BDD to create a compact representation.
Problem 3. Path-Sensitive Analysis with Satisfiability Modulo Theories.

In this problem you will use an SMT solver to find test cases exhibiting a bug in the following C function:

```c
int func(int x, int[] data, int N) {
    int v, z;
    if (0 <= x && x < N) {
        if (x >= 3) {
            x = 2 * x - 5;
        }
        v = data[x]; // ← access
        if (v >= 0 && v < N / 2) {
            z = data[2 * v]; // ← access
        } else if (v >= N/2 && v < N) {
            v = v / (x + 3);
            z = data[v]; // ← access
        } else {
            z = data[data[0]]; // ← access
        }
    }
    return z;
} else {
    return 0;
}
```

We are interested in checking whether the program can “crash” due to array out-of-bounds accesses. This means that the index with which we access the `data` array is either less than zero, or greater than N-1. To do so, first you will translate this function into an SMT formula. Then you will run an SMT solver on the formula, and interpret its output.

You may make the following assumptions about the program: N is at least 0, `data` is an array of length N, `int` refers to signed 32-bit integer, and the program terminates (crashes) as soon as the first out-of-bounds access happens.

Using SMT solvers. The research community has produced a large number of SMT solvers. For this assignment, you can use cvc5\(^1\), a state-of-the-art SMT solver used in both research and industry. You can use cvc5’s online interface to avoid the hassle of installation (link in footnote), or optionally you can download the solver binaries from its website to run locally.

To get started with writing SMT formulae, you can refer to an introductory guide\(^2\). The full SMT language specification is available at the SMT-LIB website\(^3\). For this assignment, you can use any feature defined in SMT-LIB, but the following features should be sufficient:

---

\(^1\)https://cvc5.github.io/; online interface at https://cvc5.github.io/app/
\(^2\)https://microsoft.github.io/z3guide/docs/logic/intro
\(^3\)https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf

Sorts: **Array**, **BitVec**, **Bool**

Core functions: **=**, **and**, **false**, **ite**, **not**, **or**, **true**

**Array** functions: **select**, **store**

**BitVec** functions: **bvadd**, **bvsub**, **bvmul**, **bvsdiv**, **bvsle**, **lt**, **ge**, **gt**

**Static single-assignment form.** It is easier to translate the program into a SMT formula if we can assume that every variable is defined exactly once. One way to satisfy this assumption for imperative code is by transforming the program into static single-assignment (SSA) form. To do this, first assign every variable definition a unique suffix. Then, at each join point in the control-flow graph (e.g., after branching), introduce new definitions for the variables that are defined on either path using a ϕ-node.

See the following example which transforms the imperative code on the left into SSA form on the right:

```
if (i < next) {
    if (data[i] == cookie)
        i++;
    else
        process(data[i]);
    i++;
}
if (i < next) {
    if (data[i] == cookie)
        i++;
    else
        process(data[i]);
    i++;
}
```

Follow the following steps to complete this problem.

1. **Rewrite the program in SSA form.** Notice that at different points within our function, variables v and z may refer to different definitions. Transform the function into SSA form as illustrated earlier.

2. **Translation to SMT.** The second step is to translate the program in SSA form into an SMT formula. Start your formula with the following lines:
(set-logic ALL)
(set-option :produce-models true)
(set-option :incremental true)

An assignment $x_3 = e$ becomes an assertion (assert (= x_3 E)) in SMT, where E is the translation of $e$. You need to translate int operations at the C level into bit vector operations at the SMT level. (Do not use the SMT-LIB int sort, which models unbounded integers.) For example, the translation of $x = y + 1$ is:

(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(assert (= x (bvadd y #x00000001)))
(check-sat)
(get-model)

Upon this query, an SMT solver could respond with:

sat
(
(define-fun x () (_ BitVec 32) #b00000000000000000000000000000001)
(define-fun y () (_ BitVec 32) #b00000000000000000000000000000000)
)

meaning that the SMT formula is satisfiable with model $x = 1, y = 0$. Note that the variables $x$ and $y$ are given as functions of no arguments (which must be constants because there are no side effects), and the constants themselves are given in binary or hexadecimal.

To translate arrays, use variables of the sort (Array (_ BitVec 32) (_ BitVec 32)). Array dereferences like data[i] become (select data i) when translating a read, and (store data i x) when translating a write. Note that (store data i x) returns a new array, whose $i^{th}$ element is now equal to $x$, and does not modify the original array.

To translate $\phi$-nodes, you must use a logical expression that captures the condition under which the $\phi$-node is evaluated. For example, given the following code in SSA form:

```
if (c)
    x_1 = ...;
else
    x_2 = ...;
x_3 = \phi(x_1, x_2);   // equivalent to x_3 = c ? x_1 : x_2;
```

the translation of $x_3$ is (ite c x_1 x_2). ite is short for if–then–else, and evaluates to the second or third argument based on the value of the first.

3. **Bounds checks.** The final step is to add an assertion to check each of the array accesses. You need to check that the signed value of the index is in bounds. Further, not all accesses are accessible on all paths, so you need to guard the assertion for a
particular access. The assertion should express the execution reaches this access, and it is out of bounds. Note that this will possibly constrain some path variables if the access is nested inside an if-statement, for example. For each access, you should use the sequence (push) (assert $C$) (check-sat) (pop), where $C$ is the check for that access. Push/pop allows us to add $C$ to our set of assertions temporarily, check satisfiability, and then remove it to add a different $C$. (If you added all the assertions together without push/pop, you would find a path which crash all points simultaneously, which is impossible.)

4. **Interpretation.** Once you find one or more bugs, add (get-model) after (check-sat) within the push/pop sequence for each satisfiable assertion, to print the model found for that bug.

In writing, interpret the results. Does the result indicate a crash can occur on some concrete path? If not, does this mean there can be no crash for this access? If there is a crash, translate the model into a concrete (x, data) input pair in C syntax that would crash `func()` at the corresponding access.

Hint: what happens if the program has more than one bug, and how should the model be interpreted in that case?

5. **Find the smallest value of $N$ that can cause out-of-bounds accesses in line 7.** Do this by asserting the value of $N$ in addition to asserting $C$ described in part 3. Start from $N = 1$, and increase the value of $N$ by 1 until you find a satisfying solution.

6. **Submission.** In the homework write-up, include the output of running the SMT solver on your input (sat or unsat, and, if sat, the produced model). Explain your interpretation of the results (part 4) and include answers to the questions in part 5.

Additionally, submit your SMT-LIB input file to the separate Gradescope assignment.

You should find that lines 7 and 14 are buggy, while lines 9 and 12 are safe.