# SAMPLE EXAM 1
# Testing, Debugging, and Verification
# TDA567/DIT082

---

| | |
|---|---|
| Extra aid: | Only dictionaries may be used. Other aids are *not* allowed! |
| Grade intervals: | **U**: 0 – 23p, **3**: 24 – 35p, **4**: 36 – 47p, **5**: 48 – 60p,<br>**G**: 24 – 47p, **VG**: 48 – 60p, **Max.** 60p. |

**Please observe the following:**

- This exam has 10 numbered pages.
  **Please check immediately that your copy is complete**
- Answers must be given in English
- Use page numbering on your pages
- Start every assignment on a fresh page
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions
- Indicate clearly when you make assumptions that are not given in the assignment

# Good luck!

---

**Assignment 1 (Testing)** (12p)

(a) Consider the program below. It computes the length of the longest strictly increasing subsequence in the array.
Draw a *control-flow graph* for the method `longestIncrSeq` and use it to write down a test-suite which satisfies *branch coverage* for this program.
Write your test-cases in the format `array --> result`, where `array` is an integer array and `result` is the expected result on this input. Your test-suite should be *minimal* in the sense that no two inputs should cover the same branches.

```
// pre: arr is non-null and arr.length >= 1
// post: return the length of the longest uninterrupted
// increasing sequence in arr.

public static int longestIncrSeq(int[] arr) {
 if (arr.length == 1)
   return 1;
 else {
   int i = 1;
   int count = 1;
   int maxcount = 1;
   while (i < arr.length) {
        if (arr[i - 1] < arr[i])
          count = count + 1;
        else{
          if (count > maxcount)
             maxcount = count;
          count = 1;
        }
        i = i + 1;
  }
  if (count > maxcount)
    return count;
  else
    return maxcount;
  }
}
```

Continued on next page!

(b) Another coverage criteria is *decision coverage*. Briefly explain what this is, and indicate whether or not your test-suit from part (a) satisfies this criteria as well.

(c) In addition to decision coverage, we discussed another two kinds of logic coverage in class. Describe these. Also describe the relationship between the three logic based criteria.

---

**Assignment 2 (Debugging)** (12p)

(a) When is a statement B *control dependent* on a statement A?

(b) In the small Dafny program below, on which statement(s) is/are the statements in line 9 *data dependent*? Also state why.

```
1 method M(n : nat) returns (b : nat){
2       if(n == 0)
3          { return 0; }
4       var i := 1;
5       var a := 0;
6       b := 1;
7       while (i < n)
8       {
9        a, b := b, a+b;
11       i := i +1;
12      }
13 }
```

(c) On which statements is line 11 *backward dependent*? Also state why.

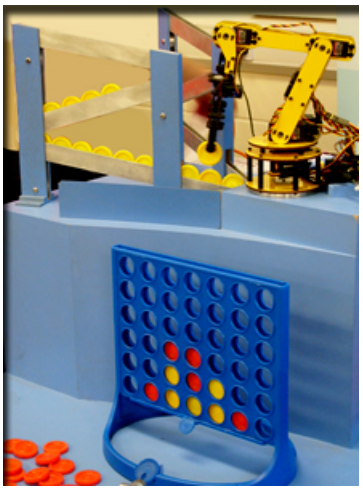
(d) The `ddMin` algorithm computes a minimal failure inducing input sequence. It relies on having a method `test(i)` which returns `PASS` if the input `i` passes the test or `FAIL` if the `i` causes failure (i.e. bug is exhibited).
Explain what we mean by *granularity* in the context of the `ddMin` algorithm.

(e) This question asks you to simulate a run of the `ddMin` algorithm. At each step, clearly state what the granularity is, how it was computed and why.
Suppose our input consists of sequences made out of the letters `A-Z`. Let `test` return `FAIL` whenever the sequence contains *two or more occurrences of the letter Z* somewhere in the sequence. The Z's does not need to be consecutive. Simulate a run of the `ddMin` algorithm and compute a minimal failing input from an initial failing input `[Z,B,R,Z,Z,Y,Z,X]`. Clearly state what happens at *each step* of the algorithm and what the final result is. Correct solutions without explanation will not be given the full score.

---

**Assignment 3 (Formal Specification)** (13p)

*Connect Four* is a two players game which takes place on a rectangular board placed vertically between them. One player has yellow tokens and the other red tokens. In each move, the player drops a token at the top of the board in one of the seven columns; the token falls down and fills the lowest unoccupied slot. Of course a player cannot drop a token in a column that is already full (i.e., it already contains six tokens). The goal is to connect four tokens vertically, horizontally, or diagonally.



We will represent the board by a two-dimensional array, called `board`. The different colours are represented by the integers 1 and 2. An empty slot is represented by the integer 0. Therefore, the above picture corresponds to the array depicted below.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 2 | 2 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 2 | 1 | 0 | 0 |
| 0 | 0 | 2 | 1 | 1 | 1 | 2 | 0 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

The game is implemented in the Dafny class `Connect4`, see below. Note that the class is generic towards the dimensions of the board, and so should your specification be. To simplify your task of specification, we omit the general winning test, and instead consider a partial winning test, `WonHorizontally`. The methods `Drop` and `WonHorizontally` are *not* robust against wrong input. Consequently, the specifications must exclude wrong inputs, using `requires` clauses.

Continued on next page!

```
class Connect4{

  // Number of columns.
  var width : int;
  // Number of rows.
  var height : int;
  // The game board.
  var board : array2<int>;
  // The filling level of all columns.
  var level : array<int>;

  predicate Valid()
  reads this, this.level;
  {}

  method Init(w : int, h : int)
  modifies this;
  {
    width := w;
    height := h;
    board := new int[w,h];
    level := new int[w];

    //Initially empty board.
    forall(i,j | 0 <= i < width && 0 <= j < height){
      board[i,j] := 0;
    }
    forall(i | 0 <= i < width) {
        level[i] := 0;
    }
  }

  method Drop(player : int, column : int)
  modifies this.level, this.board;
  {}

  method WonHorizontally(player : int) returns (won : bool)
  {}
}
```

Continued on next page!

Your task is to enrich the `Connect4` class with Dafny specifications. You are **not** required to write implementations for `Drop` and`WonHorizontally`, only specifications.

Recall that you may access the dimensions of a two-dimensional array in Dafny using `Length0` and `Length1`.

(a) Complete the body for the predicate `Valid`. It should ensure `board` and `level` have been initialised, specify what data `board` and `level` can contain and finally also how the different fields depend on each other.

(b) Write a contract specifying the `Init` function. It should check that everything is initialised as expected and that legal values have been supplied for `w` and `h`. Furthermore, it should ensure that `board` and `level` are freshly allocated objects.

(c) Write a contract for `Drop`. It should capture the assumptions that `player` is either 1 or 2, and that the column is not yet full. The method places a token of the player at the lowest free slot in that column, leaving the rest of the board unchanged.

(d) Write a contract for `WonHorizontally`. It should capture the assumption that `player` is either 1 or 2 and return true if the player has won the game with four connected tokens in a *horizontal line*. I.e. if the player has won in some other way, this method still returns false.

---

**Assignment 4 (Verification and Test Generation)** (12p)

```
method Max(arr : array<int>) returns (max : int)
requires ?
ensures ?
{
  // To be completed.
}
```

(a) Complete the above Dafny program which is supposed to compute the maximum of an array. In addition to the method body, your answer should state suitable pre- and post-conditions as well as loop invariants.

(b) **Briefly**, in concise English, explain what properties a loop invariant for *partial correctness* must satisfy.

(c) **Briefly**, in concise English, explain what *total correctness* is. What do we need, in addition to an invariant, to prove total correctness? State such an expression for the loop in the program above.

(d) When generating tests from specifications, it is normally required that the generated test inputs satisfy certain parts of the program specification. Which parts?

---

**Assignment 5 (Verification)** (11p)

This question concerns a program with a loop, which computes fibonacci numbers:

```
function fib(n : nat) : nat
{
  if (n==0) then 0 else
  if (n==1) then 1 else fib(n-1) + fib(n-2)
}

method Fib(x : nat) returns (res : nat)
ensures res == fib(x);
{
  if (x==0) { res := 0; }
  var i := 1;
  var nxt := 1;
  res := 1;
  while(i < x)
  invariant ?
  {
    res, nxt := nxt, nxt + res;
    i := i+1;
  }
}
```

(a) Give a loop *invariant* and a loop *variant* for the loop in the `Fib` method above. You may want to use the recursive function provided.

(b) Prove the `Fib` method correct using the weakest precondition calculus. You may assume that the Assignment rule works as expected for parallel assignments (i.e. as if we had written it using an intermediate variable).

---

(total 60p)