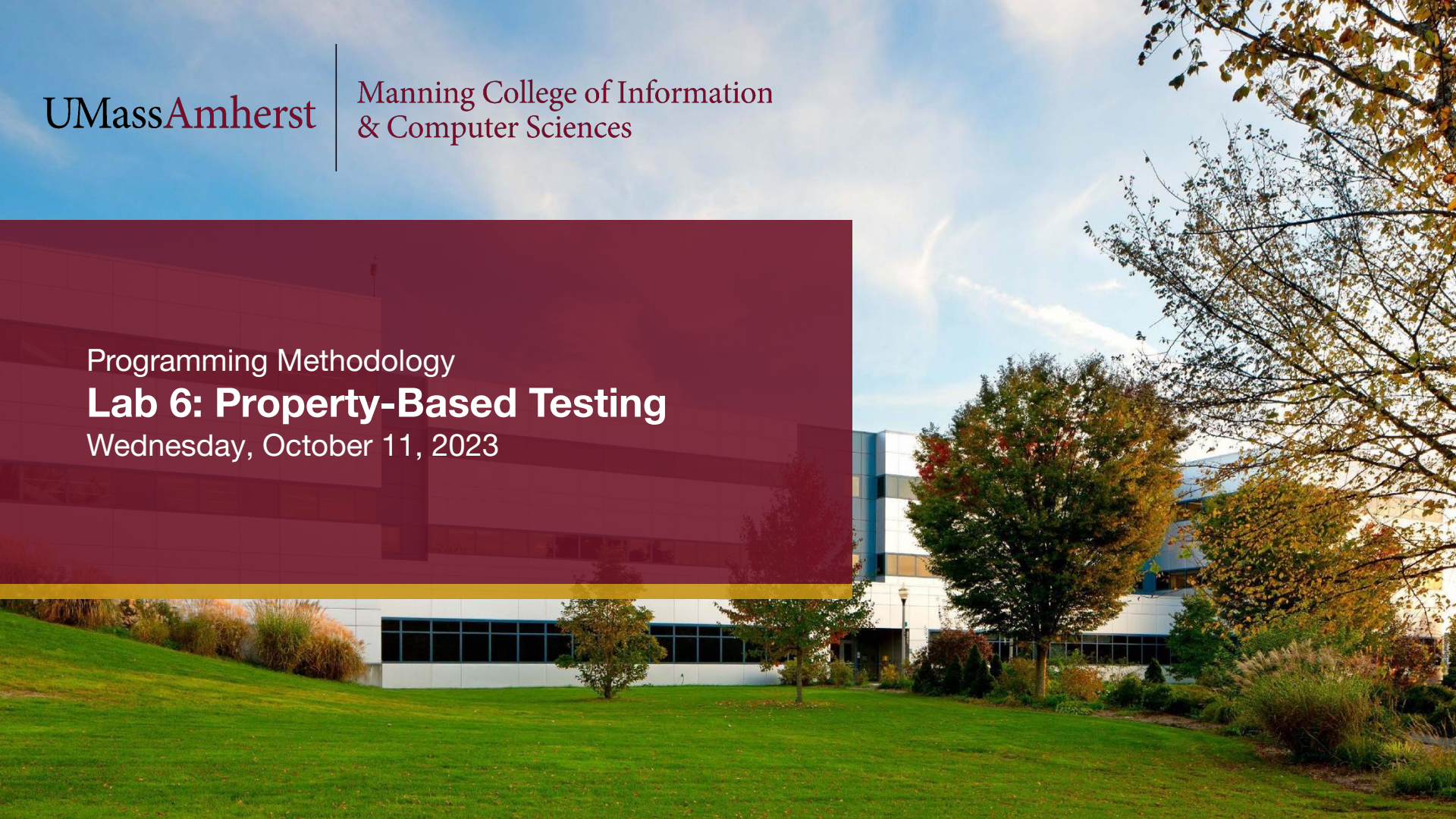


UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 6: Property-Based Testing

Wednesday, October 11, 2023

Property-Based Testing

Used when a problem has **more than one** right answer

Steps to Property-Based Testing:

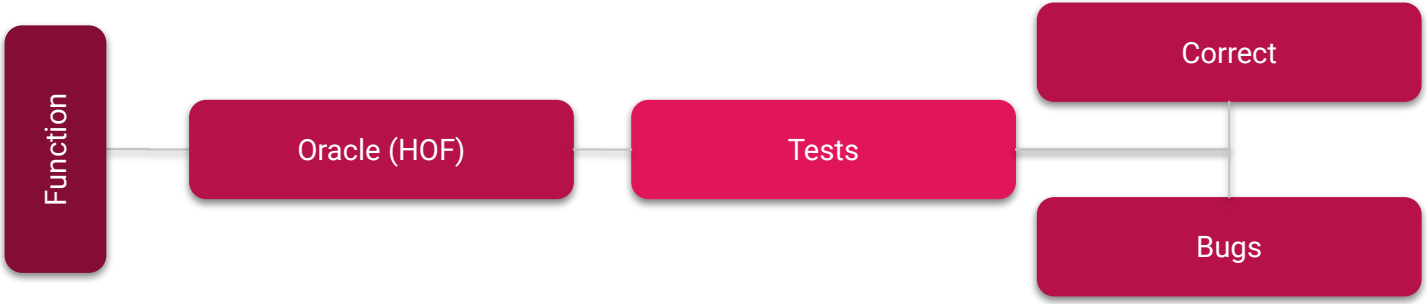
1. Start with a valid input to your problem
2. Run the algorithm on that input
3. Check that the result has all necessary characteristics

Property-Based Testing

Ex: Given a set of numbers x = {1,2,...,N-1,N}, generate a random subset of x.
What are the properties of a correct output y to this problem?

1. The number of elements in y must be less than or equal to N.
2. Every number in y must exist in x.
3. There must be no duplicates in y.

Oracle Functions



Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {

}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;

}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){

    }
}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){
        let x = arrayFrom1ToN(N); // x is {1,2,...,N-1,N}

    }
}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){
        let x = arrayFrom1ToN(N); // x is {1,2,...,N-1,N}
        let y = subsetFunc(x);

    }
}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){
        let x = arrayFrom1ToN(N); // x is {1,2,...,N-1,N}
        let y = subsetFunc(x);
        test('length of y less than or equal to N', function(){
            assert(y.length <= N);
        })
    }
}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){
        let x = arrayFrom1ToN(N); // x is {1,2,...,N-1,N}
        let y = subsetFunc(x);
        test('length of y less than or equal to N', function(){
            assert(y.length <= N);
        })
        test('all elements in y are in x', function(){
            assert(y.every(d => x.includes(d));
        })
    }
}
```

Oracle Functions

What would an oracle function for the subset function look like?

```
function oracle(subsetFunc) {
    let n = 100;
    for(let N = 0; N < n; ++N){
        let x = arrayFrom1ToN(N); // x is {1,2,...,N-1,N}
        let y = subsetFunc(x);
        test('length of y less than or equal to N', function(){
            assert(y.length <= N);
        })
        test('all elements in y are in x', function(){
            assert(y.every(d => x.includes(d));
        })
        test('y does not contain duplicates', function(){
            assert(y.every(d => (y.indexOf(d) === y.lastIndexOf(d))));
        })
    }
}
```

Exercise: Permutations

A function **genArray**(n: number): number[][] is supposed to generate an n × n array of numbers such that each row and column is a permutation of the numbers from 0 to n-1.

Write an oracle that accepts the function genArray as input.

Use higher-order functions when appropriate. Try to write your implementation in $O(n^2)$.

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {

      }

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
      }

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
            assert(p.every(valid),'valid numbers');

      }

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
            assert(p.every(valid),'valid numbers');
            const f = Array(n).fill(1);
            assert(p.every(e => --f[e] === 0), 'no duplicates');
      }

}
```

Note: We're checking for duplicates in a single linear scan which takes O(n) time.
This approach works for n = 0 and an empty array.

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
            assert(p.every(valid),'valid numbers');
            const f = Array(n).fill(1);
            assert(p.every(e => --f[e] === 0), 'no duplicates');
      }
      assert(a.length === n, 'array has n rows');

}
```

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
            assert(p.every(valid),'valid numbers');
            const f = Array(n).fill(1);
            assert(p.every(e => --f[e] === 0), 'no duplicates');
      }
      assert(a.length === n, 'array has n rows');
      a.forEach(isPerm); // each row is a permutation

}
```

a is length n, and we call isPerm (an O(n) function) for each element of a. This will take $O(n^2)$ time

Solution: Permutations

Write an oracle that accepts the function genArray as input.

```
function oracle(genArray) {
      let n = Math.floor(Math.random()*1000);
      let a = genArray(n);
      const valid = x => x % 1 === 0 && 0 <= x && x < n;
      function isPerm(p) {
            assert(p.length === n, 'length is n');
            assert(p.every(valid),'valid numbers');
            const f = Array(n).fill(1);
            assert(p.every(e => --f[e] === 0), 'no duplicates');
      }
      assert(a.length === n, 'array has n rows');
      a.forEach(isPerm); // each row is a permutation
      for (let k = 0; k < n; ++k) { isPerm(a.map(r => r[k])); } // each column is a permutation
}
```

Again, this will take $O(n^2)$ time

Note: A different approach, where we check directly that every number from 0 to n – 1 is included requires a linear scan for each number. That would be $O(n^2)$ per row/column and $O(n^3)$ overall.

Review Exam Questions

With the left over time we'll go over exam questions that you all found difficult.

Suggestion: Q3 or Q4, but we can go over any you want.

Exercise 2: OOP

Define an iterator class over an array.
Define a class Polygon (defined by an array of points in 2D) which has a makeIterator() method for its vertices.
Extend the class to CenteredPolygon which has a method to compute its center.
The center of weight has the average of the vertex coordinates.

```
class ArrayIterator<T> implements IterableIterator<T> {.......}
```

```
class Polygon {.......}
```

```
class CenteredPolygon extends Polygon {.......}
```

Exercise 2: Solution


```
class ArrayIterator<T> implements IterableIterator<T> {
    private array: T[];
    private index = 0;
    constructor(private array: T[]) {this.array = array}
    public next(): IteratorResult<T> {
        if (this.index < this.array.length) {
        return this.array[this.index++];
        } else {
        return null;
        }
    }
}
```

```
  class Polygon {
    private vertices: [number,number][];
    constructor(private vertices:[number, number][]) {
            this.vertices = vertices;
      }

    public makeIterator(): IterableIterator<number[]>
      {
      return new ArrayIterator<number[]>(this.vertices);
    }
  }
```

```
class CenteredPolygon extends Polygon {
    public Center(): number[] {

        const numVertices = this.vertices.length;

        let x = 0;
        let y = 0;
        const iterator = this.makeIterator();
        let current = iterator.next();

        while (current !== null) {
            x += current[0];
            y += current[1];
            current = iterator.next();
        }


        return [x / numVertices, y / numVertices];
    }
}
```

Exercise 2: Solution 2


```
interface Iterator<T> {
    next(): { value: T, done: boolean };
}
```

```
class Polygon {
    private vertices:[number, number][];

    constructor(vertices:[number, number][]) {
      this.vertices = vertices;
    }

    makeIterator(): Iterator<number[]> {
      let index = 0;
      const vertices = this.vertices;

      return {
        next() {
          if (index < vertices.length) {
            return { value: vertices[index++], done: false };
          } else {
            return { value: null, done: true };
          }}};}}
```

```
class CenteredPolygon extends Polygon {
    private readonly sides: number;

    constructor(vertices: number[][]) {
      super(vertices);
      this.sides = vertices.length;
    }
    const numVertices = this.vertices.length;
    center(): number[] {
      let x = 0;
      let y = 0;
      const iterator = this.makeIterator();
      let current = iterator.next().value;

      while (current !== null) {
        x += current[0];
        y += current[1];
        current = iterator.next().value;
      }

      return [x / this.sides, y / this.sides];
    }
  }
```

Exercise 2: Solution 3


```
class ArrayIterator<T> {
  private arr: T[];
  private idx: number;
  constructor(a: T[]) { this.arr = a; this.idx = -1;
}
  next(): T | null {
    return ++this.idx < this.arr.length ?
                    this.arr[this.idx] : null;
  }
}
```

```
class Polygon {
  private vertices: [number,number][];
  constructor(vertices: [number,number][]) {
    this.vertices = vertices;
  }
  makeIterator() {
    return new ArrayIterator(this.vertices);
  }
}
```

```
class CenteredPolygon extends Polygon {
  center() {
    const it = this.makeIterator();
    let xs = 0, ys = 0, cnt = 0;
    for (let v: [number, number] | null;
          (v = it.next()) != null; ++cnt) {
      xs += v[0]; ys += v[1];
    }
    return [xs/cnt, ys/cnt];
  }
}
```