UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 9: Midterm 2 Review

Wednesday, October 29th, 2025

Reminders

- Midterm 2 is tonight, 7-9pm!
  - ISB 135: Discussion labs LL, LM, LN, LQ
  - Morrill2 131: Discussion labs LR LS, LU, LV
  - Advice:
    - Write legibly (reduces risk of misgrading)
    - Write the types of function parameters and return values
    - The order of the questions doesn't mean anything, feel free to do them in whatever order is best for you
- HW 5 is due this Sunday (11/2) at 11:59pm
  - Come to office hours for help!

UMassAmherst | Manning College of Information & Computer Sciences

Today’s Goals

- Review midterm 2 material
  - Property-based Testing
  - Object-Oriented Programming
  - Iterators
  - Generators

Exercise 1: Property-based Testing

Write property-based tests for the function **`partition(arr: number[], p: number)`** with the following specification:

- The array elements and **`p`** are assumed to be integers between 0 and **`arr.length-1`**.
- The function rearranges the array in place, so that: a number with value > **`p`** will not appear before a number with value <= **`p`**, and a number with value < **`p`** will not appear after a number with value >= **`p`**.

Try to write a complete set of tests, adding comments to clearly indicate the purpose of each test.

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length.
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before calling **`partition`** and comparing.
3. Check that the result has elements less than **`p`**, then elements equal to **`p`**, then elements greater than **`p`** (any of those may be missing).

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length.

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  // ...
}
```

We will write a helper function `checkSame` to test for the first two tests we have described.

We begin by simply checking if the two arrays have the same length.

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length.
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before calling partition and comparing.

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  const isValid = (e: number) => 0 <= e && e < a.length && e % 1 === 0;
  let counts = new Array<number>(a.length).fill(0);
  // ...
}
```

Next, we define a helper function `inRange` to determine if a number is an integer between 0 and the length of the array minus one, as the problem specifies.

We then create an array counts to track the number of times each number appears, initializing all counts to 0.

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length.
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before calling partition and comparing.

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  const isValid = (e: number) => 0 <= e && e < a.length && e % 1 === 0;
  let counts = new Array<number>(a.length).fill(0);
  b.forEach(e => {
    assert(isValid(e));
    ++counts[e];
  });
  // ...
}
```

We use forEach to check that each element in the array b (result from partition) is valid. We then also update our array of counts, tracking the number of times each value appears in b.

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length.
2. Check that they contain the same elements with the same frequencies. Together, this can be done by making a copy before calling partition and comparing.

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  const isValid = (e: number) => 0 <= e && e < a.length && e % 1 === 0;
  let counts = new Array<number>(a.length).fill(0);
  b.forEach(e => {
    assert(isValid(e), “Values should be an integer in the correct range”);
    ++counts[e];
  });
  a.forEach(e => assert(--counts[e] >= 0, “Values should appear at same frequency”));
}
```

Lastly, we call forEach on the original array (a) and ensures that each element does not appear more often in the original array (a) than it does in the new array (b).

Exercise 1: Solution

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  const isValid = (e: number) => 0 <= e && e < a.length && e % 1 === 0;
  let counts = new Array<number>(a.length).fill(0);
  b.forEach(e => {
    assert(isValid(e), "Values should be an integer in the correct range");
    ++counts[e];
  });
  a.forEach(e => assert(--counts[e] >= 0, "Values should appear at same frequency"));
}
```

Is the last line sufficient to check that the elements have the same frequencies in the two arrays?

Yes; the last check ensures that each element in a does not appear more often than it does in b. Now, if there was an element of b that appears more often in b than in a, then the array counts should have at least one non-zero count after the last line.

But this would require the array b to be longer than the array a, which we know is not true.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length
2. Check that they contain the same elements with the same frequencies.
   Together, this can be done by making a copy before partition and comparing:
3. Check that the result has elements less than **p**, then elements equal to **p**, then elements greater than **p** (any of those may be missing).

```
function checkPivot(a: number[], p: number): void {
  let i = a.length; if (i === 0) { return; }
  while (i > 0 && a[--i] > p) {} //all > p
  while (i >= 0 && a[i] === p) { --i; } //all = p
  while (i >= 0) { assert(a[i] < p); --i; }
}
```

For the third test, we can iterate through the resulting array from right to left. We use while loops to first iterate through all elements greater than **p**, then all elements equal to **p**, and lastly assert that all remaining elements are less than **p**.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 1: Solution

Tests:

1. Check that the original and resulting array have the same length
2. Check that they contain the same elements with the same frequencies.
   Together, this can be done by making a copy before partition and comparing:
3. Check that the result has elements less than **p**, then elements equal to **p**, then elements greater than **p** (any of those may be missing).

```
function checkSame(a: number[], b: number[]): void {
  assert(a.length === b.length);
  const isValid = (e: number) =>
      0 <= e && e < a.length && e%1 === 0
  let f = new Array<number>(a.length).fill(0);
  b.forEach(e => { assert(inRange(e)); ++f[e]; });
  a.forEach(e => assert(--f[e] >= 0));
}
```

```
function checkPivot(a: number[], p: number): void {
  let i = a.length; if (i === 0) { return; }
  while (i > 0 && a[--i] > p) {} //all > p
  while (i >= 0 && a[i] === p) { --i; } //all = p
  while (i >= 0) { assert(a[i] < p); --i; }
}
```

You would then use these two functions to test the output of any given partition function to check if it follows the specifications.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2: OOP

Write a class **CandyInventory** with the following public methods (and no other accessible attributes):

- `constructor(preferredCandy: string[])`
    - Takes in an array of candy names, ranked by preference.
    - i.e. the first string is the most preferred candy (rank 1), etc.
- `addCandy(r: number, amount: number): void`
    - Adds an amount of the candy with preference rank r to the inventory. Both arguments are assumed to be positive integers. No changes should be made if there is no candy of rank r.
- `makeIterator(threshold: number): () => Iterator<number>`
    - Takes a threshold as input. Returns a closure that returns an iterator. The iterator should yield the names of all candy with a quantity greater than the given threshold, in order of highest to lowest preference.
    - Use `[Symbol.iterator]()`.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2: Solution

```
class CandyInventory {
    private candyNames: string[];
    private candyAmounts: number[];

    constructor(preferredCandy: string[]) {
        this.candyNames = preferredCandy;
        this.candyAmounts = Array(this.candyNames.length).fill(0);
    }

    // ...
}
```

We store all necessary information with two attributes, one to store the names of the candy (ordered by preference) and one to store the corresponding amounts of each candy.

In the constructor, we initialize the amount we have of each candy to zero.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2: Solution

```
class CandyInventory {
    private candyNames: string[];
    private candyAmounts: number[];

    addCandy(r: number, amount: number): void {
        if (r <= this.candyNames.length) {
            this.candyAmounts[r-1] += amount;
        }
    }

    // ...
}
```

For **addCandy**, we first check if **r** is a valid rank. Since we are told it is a positive integer, we only need to check to see if the rank is too high. Since ranks start at 1, this means **r** should be no greater than the number of candy names we have.

If **r** is a valid rank, we then increment the amount of that candy. We subtract by 1 to account for ranks starting at 1.

Exercise 2: Solution

```
class CandyInventory {
    private candyNames: string[];
    private candyAmounts: number[];

    makeIterator(threshold: number): () => Iterator<string> {
        let i = 0;
        const filteredCandy = this.candyNames.filter(_ => {
            const aboveThreshold = this.candyAmounts[i] > threshold;
            i++;
            return aboveThreshold;
        })
        // ...
    }
}
```

We first filter **this.candyNames**, creating a new array **filteredCandy** with the candy that is above the given threshold. Instead of using each candy name, we keep track of the current index in the array in order to find the corresponding value in **this.candyAmount** for the current candy.

Exercise 2: Solution

```
class CandyInventory {
    private candyNames: string[];
    private candyAmounts: number[];

    makeIterator(threshold: number): () => Iterator<string> {
        let i = 0;
        const filteredCandy = this.candyNames.filter(_ => {
            const aboveThreshold = this.candyAmounts[i] > threshold;
            i++;
            return aboveThreshold;
        })
        return () => filteredCandy[Symbol.iterator]();
    }
}
```

Lastly, we return a closure that uses **[Symbol.iterator]()** to create a new iterator each time it is called.

Exercise 3: Iterators

Write a function **`makeWindowSums`** that takes an **`Iterable<number>`**, and returns an **`Iterator<number>`**. The returned iterator should return the sum of every 3 consecutive numbers produced by the iterable.

Specifically, suppose the given iterable produces the sequence $(a_1, a_2, \ldots, a_n)$, where **n** is the total number of values produced by the iterable. Then, the i-th value yielded by the iterator should be equal to $a_i + a_{i+1} + a_{i+2}$. If any of these values go past the end of the sequence $(a_1, a_2, \ldots, a_n)$, your iterator should terminate.

You should design a class `WindowSumsIterator` that implements the Iterator interface, and use this class in your implementation of `makeWindowSums`. Do not store the entire iterable as an array.

Exercise 3: Solution

```
class WindowSumsIterator implements Iterator<number> {
    private iterator: Iterator<number>;
    private item1: IteratorResult<number>;
    private item2: IteratorResult<number>;

    // ...
```

We begin by defining three private attributes. The **iterator** attribute will allow us to store an iterator for the given iterable, and the item attributes will store the previous two IteratorResults from **this.iterator**.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 3: Solution

```
class WindowSumsIterator implements Iterator<number> {
    private iterator: Iterator<number>;
    private item1: IteratorResult<number>;
    private item2: IteratorResult<number>;

    constructor(iterable: Iterable<number>) {
        this.iterator = iterable[Symbol.iterator]();
        this.item1 = this.iterator.next();
        this.item2 = this.iterator.next();
    }

    // ...
```

For the constructor, we take an iterable as input and obtain an iterator for it using the **[Symbol.iterator]()** method. We then use **.next()** to get the first two items, storing them in their respective attributes.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 3: Solution

```
class WindowSumsIterator implements Iterator<number> {
    private iterator: Iterator<number>;
    private item1: IteratorResult<number>;
    private item2: IteratorResult<number>;

    next(): IteratorResult<number> {
        const curr = this.iterator.next();
        if (curr.done) {
            return {done: true, value: undefined};
        }
        const sum = this.item1.value + this.item2.value + curr.value;
        // ...
    }
}
```

We first obtain the current item from the iterator. If this **IteratorResult** indicates that the iterator is terminated, then our iterator has also finished and we can return the **IteratorResult** indicating this. Otherwise, we calculate the sum of the previous 2 values and the current value.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 3: Solution

```
class WindowSumsIterator implements Iterator<number> {
    private iterator: Iterator<number>;
    private item1: IteratorResult<number>;
    private item2: IteratorResult<number>;

    next(): IteratorResult<number> {
        const curr = this.iterator.next();
        if (curr.done) {
            return {done: true, value: undefined};
        }
        const sum = this.item1.value + this.item2.value + curr.value;
        this.item1 = this.item2;
        this.item2 = curr;
        return {done: false, value: sum};
    }
}
```

We then set up **this.item1** and **this.item2** for future calls to **next()**. We must store curr as a previous item now, so we will move **this.item2** into **this.item1**. We then return an **IteratorResult**.

Exercise 3: Solution

```
function makeWindowSums(iterable: Iterable<number>): Iterator<number> {
    return new WindowSumsIterator(iterable);
}
```

Lastly, we use the class we defined in the makeWindowSums function, returning a new instance of the WindowSumsIterator class.

Exercise 4: Generators

Write a generator function minimaGenerator that takes an Iterator<number> as input and returns a Generator<number>. The generator should yield all the minima in the sequence of numbers returned by the iterator in their original order. A number x is a minimum if a number exists before and after it in the sequence, and x is strictly less than both of these two numbers.

For example, if the given iterator yields the sequence (3, 2, 4, 1, 5, 0), then your generator should yield the sequence (2, 1).

Exercise 4: Solution

```
function* minimaGenerator(iterator: Iterator<number>): Generator<number> {
    let prev = iterator.next();
    let curr = iterator.next();
    while (!curr.done) {
        // ...
    }
}
```

We begin by initializing two variables to track the previous and current values from the iterator. We will also need to track a third value (the next value) to determine if **curr** is a minima, but this will be done within the while loop.

Exercise 4: Solution

```
function* minimaGenerator(iterator: Iterator<number>): Generator<number> {
    let prev = iterator.next();
    let curr = iterator.next();
    while (!curr.done) {
        let next = iterator.next();
        if (!next.done && curr.value < Math.min(prev.value, next.value)) {
            yield curr.value;
        }
        // ...
    }
}
```

We obtain the next item from the iterator. If the iterator is not finished yet, then **next** will be an **IteratorResult** with a value.

After we check that **next.done** is false, we check if our **curr.value** is a minimum by seeing if it is strictly less than **prev.value** and **next.value**. This is equivalent to comparing **curr.value** to the minimum of the two. We yield the current value if it is a minimum.

UMassAmherst | Manning College of Information & Computer Sciences

Exercise 4: Solution

```
function* minimaGenerator(iterator: Iterator<number>): Generator<number> {
    let prev = iterator.next();
    let curr = iterator.next();
    while (!curr.done) {
        let next = iterator.next();
        if (!next.done && curr.value < Math.min(prev.value, next.value)) {
            yield curr.value;
        }
        prev = curr;
        curr = next;
    }
}
```

Lastly, we update **prev** and **curr** for the next iteration. **curr** moves back and becomes **prev**, and **next** is stored as **curr**.