UMass Amherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 13: Final Review

Wednesday, May 7th, 2025

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Lab leaders will assign new groups this week
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Great job this semester everyone, you should be proud of your hard work!
- Office hours will continue to happen as scheduled.
  - Exceptions will be announced on campuswire.
- Exam Logistics:
  - Wednesday May 13th 6pm - 8pm in ILC N151.
  - Make sure to check SPIRE on exam day in case there is a last minute location change!
- Please fill out the SRTI course survey, this really helps make the class better!
- HW9 is due tomorrow midnight
- HW8 Self Eval on gradescope is due Friday at midnight

Today's Goals

- Practice working with program correctness
- Practice working with asynchronous programming

Exercise: Program Correctness

The following code should partition the given array in-place such that all odd numbers come before all even numbers.

First, write the invariants which satisfy the high-level algorithm.

Then, fill in the code to satisfy the invariants.

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  // low/high form a window, the outside of which is partitioned;
  // the window shrinks iteratively until everything is partitioned
  while (???) {
    if (???) {
      // swap arr[low] and arr[high]
      ???
    }
    if (???) {
      // update low
      ???
    }
    if (???) {
      // update high
      ???
    }
  }
}
```

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  while (???) {
    // low/high form a window, the outside of which is partitioned
    // => (everything before low is odd) and (everything above high is even)
    // => (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
    if (???) {
      // swap arr[low] and arr[high]
      ???
    }
    if (???) {
      // update low
      ???
    }
    if (???) {
      // update high
      ???
    }
  }
}
```

write a loop invariant; we can take our intuitive understanding the algorithm, write it down in plain english, then iteratively refine it until it is more mathematically formal

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  // let Inv be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // Inv
  while (???) {
    // Inv && ???
    if (???) { // arr[low], arr[high] are of the opposite parity needed
      // Inv && ???
      // swap arr[low] and arr[high]
      // ???
    } else { // empty else, do nothing, sa
    } // establish joint conclusion
    // ???
    if (???) {
      // ???
      // update low
      ???
      // ???
    }
    // ???
    if (???) {
      // ???
      // update high
      ???
      // ???
    }
    // ???
  }
  // Inv should be restored here
}
```

At various points in the code, the invariant might hold with extra conditions, or might not hold, but be restored later we now have enough information to fill in initialization; there is only one option which satisfies the invariant

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let Inv be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // Inv
  while (???) {
    // Inv
    if (arr[low] % 2 === 0 && arr[high] % 2 === 1) {
      // Inv && arr[low] is even and arr[high] is odd
      [arr[low], arr[high]] = [arr[high], arr[low]];
      // Inv && arr[low] is odd and arr[high] is even
    } else { // arr[low] is odd || arr[high] is even (negated condition
      // Inv && (arr[low] is odd || arr[high] is even)
    } // condition on else branch includes condition on then branch
    // Inv && (arr[low] is odd || arr[high] is even)
    if (arr[low] % 2 === 1) {
      // Inv and (arr[low] is odd)
      // (forall i: (i < low) => (arr[i] is odd)) and (arr[low] is odd) => (forall i: (i < low+1) => (arr[i] is odd))
      low += 1;
      // Inv (assignment rule gives forall i: (i < low) => (arr[i] is odd) restoring invariant
      // high - low = \old(high) - \old(low) - 1
    } // else Inv && arr[high] is even
    // Inv && (high - low = \old(high - low) - 1 || arr[high] is even)
    if (arr[high] % 2 === 0) {
      // Inv and (arr[high] is even)
      // (forall i: (i > high) => (arr[i] even)) and (arr[high] even) => (forall i: (i > high - 1) => (arr[i] is even))
      high -= 1;
      // Inv (assignment rule gives forall i: (i > high) => (arr[i] even) restoring invariant
      // high - low = \old(high) - 1 - \old(low)
    } // else Inv && high - low = \old(high - low) - 1
    // Inv && high - low = \old(high - low) - 1: invariant restored + progress made
```

we know how to swap two elements; this does not violate the invariant

let's also track progress: does high - low decrease?

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let Inv be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // Inv
  while (???) {
    // Inv
    if (arr[low] % 2 === 0 && arr[high] % 2 === 1) {
      // Inv && arr[low] is even and arr[high] is odd
      [arr[low], arr[high]] = [arr[high], arr[low]];
      // Inv && arr[low] is odd and arr[high] is even
    } // else Inv && (arr[low] is odd || arr[high] is even)
    // Inv && (arr[low] is odd || arr[high] is even)
    if (arr[low] % 2 === 1) { // Inv and (arr[low] is odd)
      // (forall i: (i < low) => (arr[i] is odd)) and (arr[low] is odd) => (forall i: (i < low+1) => (arr[i] is odd))
      low += 1;
      // Inv && high - low = \old(high) - \old(low) - 1
    } // else Inv && arr[high] is even
    // Inv && (high - low = \old(high - low) - 1 || arr[high] is even)
    if (arr[high] % 2 === 0) { // Inv and (arr[high] is even)
      // (forall i: (i > high) => (arr[i] even)) and (arr[high] even) => (forall i: (i > high - 1) => (arr[i] is even))
      high -= 1;
      // Inv && high - low = \old(high) - 1 - \old(low)
    } // else Inv && high - low = \old(high - low) - 1
    // Inv && high - low = \old(high - low) - 1
  }
  // Inv and (???)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall j: (i < j) => (arr[j] is even)))
}
```

to write the “while” condition, we need to know the desired state when the loop finishes; we want the array to be fully partitioned

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let Inv be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // Inv
  while (low <= high) {
    // Inv
    if (arr[low] % 2 === 0 && arr[high] % 2 === 1) {
      // Inv && arr[low] is even and arr[high] is odd
      [arr[low], arr[high]] = [arr[high], arr[low]];
      // Inv && arr[low] is odd and arr[high] is even
    } // else Inv && (arr[low] is odd || arr[high] is even)
    // Inv && (arr[low] is odd || arr[high] is even)
    if (arr[low] % 2 === 1) { // Inv and (arr[low] is odd)
      // (forall i: (i < low) => (arr[i] is odd)) and (arr[low] is odd) => (forall i: (i < low+1) => (arr[i] is odd))
      low += 1;
      // Inv && high - low = \old(high) - \old(low) - 1
    } // else Inv && arr[high] is even
    // Inv && (high - low = \old(high - low) - 1 || arr[high] is even)
    if (arr[high] % 2 === 0) { // Inv and (arr[high] is even)
      // (forall i: (i > high) => (arr[i] even)) and (arr[high] even) => (forall i: (i > high - 1) => (arr[i] is even))
      high -= 1;
      // Inv && high - low = \old(high) - 1 - \old(low)
    } // else Inv && high - low = \old(high - low) - 1
    // Inv && high - low = \old(high - low) - 1
  }
  // Inv and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall j: (i < j) => (arr[j] is even)))
}
```

once we've figured out what should hold after the loop is over, then the "while condition" will just be the negation of that

we've established both termination and the desired outcome

Exercise 2: Async

Write a function asyncPosMap(*arr*: T[], *f*: T => Promise(number)): Promise<T[]>. This function takes a generic array, *arr* and an asynchronous function *f*, and returns a new Promise. That promise should be fulfilled with a new array containing the elements of *arr* that for which f resolved to a positive number. The promise should reject if at any point *f* rejects. Ensure that calls to f occur asynchronously, by using Promise.all.

Fall 2022 Midterm 2 Makeup - 20pts

Exercise 2: Solution

UMassAmherst | Manning College of Information & Computer Sciences

```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {

    // The first thing we have to do is get the result of applying f to every element in arr.

}
```

Exercise 2: Solution

UMassAmherst | Manning College of Information & Computer Sciences

```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
        return Promise.all(                );

        // To do this we'll use Promise.all. Why do we need to do this?





}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
    return Promise.all(arr.map(f))

    // Let's think about what arr.map(f) will return. It returns an array of promises that will
    // resolve to a number, i.e., Promise<number>[].
    // It would be a lot more convenient if we had a Promise that resolved to a number[]
    // when all the asynchronous functions return.
    // This is what Promise.all will do.
    // It will transform our Promise<number>[] into a Promise<number[]>.
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
	return Promise.all(arr.map(f)).then(

		// Now we'll use a .then to filter and retain the elements for which f resolved
		// to a positive number.
		// Notice that we can use .then because Promise.all gave us a single promise
		// instead of an array of promises. (Yay)



	);
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
	return Promise.all(arr.map(f)).then(
		(nums: number[]) => {

			// Since our promise resolves to a number[], let's use it now to filter.



		}
	);
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
	return Promise.all(arr.map(f)).then(
		(nums: number[]) => {
			// need to return elements of arr, not nums
			// Return type is Promise<T[]>, not Promise<number[]>
			return arr.filter(...);
		}
	);
}
```

Exercise 2: Solution

UMassAmherst | Manning College of Information & Computer Sciences

```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
	return Promise.all(arr.map(f)).then(
		(nums: number[]) => {
			// .filter (and other array HoF) can supply index of the current element
			// we don’t need the element (first parameter) so we use “_”
			// what condition replaces …?
			return arr.filter((_, i) => … );
		}
	);
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
	return Promise.all(arr.map(f)).then(
		(nums: number[]) => {
			return arr.filter((_, i) => nums[i] > 0 );
		}
	);
}

// If any call to f rejects, Promise.all will return a promise that rejects. This will bubble
// though our .then. Thus, if f rejects, so does our returned promise.
```

Anonymous Lab Feedback

UMassAmherst | Manning College of Information & Computer Sciences

https://forms.gle/5FGu6nCx9xKFFVVEA