CS 61BL Summer 2026

Arrays, Linked Lists, and ADTs

Recitation 01: July 6th, 2026

1 Static Electricity

Consider the class `Pokemon`, implemented as shown below:

```
public class Pokemon {
    public String name;
    public int level;
    public static String trainer = "Ash";
    public static int partySize = 0;

    public Pokemon(String name, int level) {
        this.name = name;
        this.level = level;
        this.partySize += 1;
    }

    public static void main(String[] args) {
        Pokemon p = new Pokemon("Pikachu", 17);
        Pokemon j = new Pokemon("Jolteon", 99);
        System.out.println("Party size: " + Pokemon.partySize);
        p.printStats();
        int level = 18;
        Pokemon.change(p, level);
        p.printStats();
        Pokemon.trainer = "Ash";
        j.trainer = "Cynthia";
        p.printStats();
    }

    public static void change(Pokemon poke, int level) {
        poke.level = level;
        level = 50;
        poke = new Pokemon("Luxray", 1);
        poke.trainer = "Team Rocket";
    }

    public void printStats() {
        System.out.println(name + " " + level + " " + trainer);
    }
}
```

(a) Write what would be printed after the `main` method is executed.

(b) On line 28, we set `level` equal to `50`. What `level` do we mean?

A. An instance variable of the `Pokemon` object
B. The local variable containing the parameter to the `change` method
C. The local variable in the `main`
D. Something else (explain below)

(c) If we were to call `Pokemon.printStats()` at the end of our main method, what would happen?

2 Interweave

Implement **interweave**, which takes in an **IntList lst** and an integer **k**, and *destructively* interweaves **lst** into **k IntList**s, stored in an array of **IntList**s. Here, destructively means that instead of creating new **IntList** instances, you should focus on modifying the pointers in the existing **IntList lst**.

Specifically, we require:

- It is the **same** length as the other lists. You may assume the **IntList** is evenly divisible.
- The first element in **lst** is put in the first index of the array of **IntList**s. The second element is put in the second index. This goes on until the array is traversed, and then we wrap around to put elements in the first index of the array.
- Its ordering is consistent with the ordering of **lst**, i.e. items in earlier in **lst** must **precede** items that are later.

For instance, if **lst** contains the elements [6, 5, 4, 3, 2, 1], and **k** = 2, then the method should return an array of **IntList**, [6, 4, 2] at index 0, and [5, 3, 1] at index 1.

In the beginning, we reversed the **IntList lst** destructively, because it's usually easier to build **IntList** backwards.

Hint: The elements in the array should track the head of the small **IntList** that they are building.

```
public static IntList[] interweave(IntList lst, int k) {
  IntList[] array = new IntList[k];
  int index = k - 1;
  IntList L = reverse(lst); // Assume reverse is implemented correctly

  while (_____) {

    IntList prevAtIndex = _____;

    IntList next = _____;

    _____;

    _____;

    L = _____;

    index -= 1;

    if (_____) {

      _____;

    }
  }
  return array;
}
```

3 OHQueue

Dawn is designing the new 61B Office Hours Queue. The code below for OHRequest represents a single request.

```
public class OHRequest {
    public String description;
    public String name;
    public boolean isSetup;
    public OHRequest next;
    public OHRequest(String description, String name, boolean isSetup, OHRequest next) {
        /* Omitted: The constructor sets the instance variables that
        have the corresponding names. */
    }
}
```

Dawn would like to find a way to prioritize setup tickets on the queue so that they appear at the top. She wants to implement this based on the isSetup field of each OHRequest, but sometimes students forget to set it to true, so she decides to use description as backup to break ties.

(a) Fill in the compare method of OHRequestComparator below. First, if one but not both of the OHRequests have their isSetup set to true, the one with isSetup set to true should take priority (ie. earlier on the queue). If both or neither of the OHRequests have their isSetup set to true, tiebreak with the description: the description has to exactly match "setup" in order to be counted as a setup issue. If both requests have such descriptions, it's a true tie and return 0.

Note: In this question, **we define s1 being "less than" s2 as s1 being prioritized over s2.** Consider it equivalent to comparing the two requests' position in the queue (1 vs 3, for instance).

```
public class OHRequestComparator implements Comparator<_____> {
    @Override
    public int compare(_____ s1, _____ s2) {
        boolean s1DescMatchesSetup = s1.description.equals(_____);
        boolean s2DescMatchesSetup = s2.description.equals(_____);
        if (_____) {
            return -1;
        } else if (_____) {
            return 1;
        } else if (_____) {
            return -1;
        } else if (_____) {
            return 1;
        }
        return 0;
    }
}
```

(b) Create a class `OHIterator` that implements an `Iterator` over `OHRequests` and only returns requests with good descriptions (using the `isGood` function). Our `OHIterator`'s constructor takes in an `OHRequest` that represents the first `OHRequest` on the queue. If we run out of requests when we try to get the next one, throw a `NoSuchElementException` using `throw new NoSuchElementException();`

```
public class OHIterator implements Iterator<OHRequest> {
  OHRequest curr;
  public OHIterator(OHRequest queue) {

    _____________________________________________;
  }
  public static boolean isGood(String description) {
    return description.length() >= 5;
  }
  @Override
  public boolean hasNext() {

    while (_____________________________ && _____________________________) {

      _____________________________;
    }
    if (curr == null) {
      return false;
    }
    return true;
  }
  @Override
  public OHRequest next() {
    if (!hasNext()) {

      throw new _____________________________;
    }

    OHRequest currRequest = _____________________________;

    _____________________________;

    _____________________________;
  }
}
```

4 Gridify

(a) Consider a circular sentinel implementation of an **SLList** of **Node**s. For the first **rows** * **cols** **Node**s, place the item of each **Node** into a 2D **rows** × **cols** array in row-major order. Elements are sequentially added filling up an entire row before moving onto the next row.

For example, if the **SLList** contains elements $5 \to 3 \to 7 \to 2 \to 8$ and **rows** $= 2$ and **cols** $= 3$, calling **gridify** on it should return this grid.

5	3	7
2	8	0

If the SLList contains fewer elements than the capacity of the 2D array, the remaining array elements should be 0; if it contains more elements, ignore the extra elements.

```
public class SLList {
  Node sentinel;

  public SLList() {
    this.sentinel = new Node();
  }

  private static class Node {
    int item;
    Node next;
  }

  public int[][] gridify(int rows, int cols) {

    int[][] grid = _____;

    _____;

    return grid;
  }

  private void gridifyHelper(int[][] grid, Node curr, int numFilled) {

    if (_____) {

      return;
    }


    int row = _____;

    int col = _____;


    grid[row][col] = _____;

    _____;

  }
}
```

5 In Circles 2: More Circles

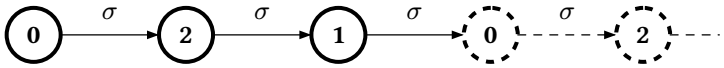
In abstract algebra, an **orbit** is defined as a closed path of elements generated by a permutation of a set. A permutation is a function that takes a set and **rearranges some or all of its elements**.

For example, given the following permutation σ of the set of integers in the interval $[0, 5]$,

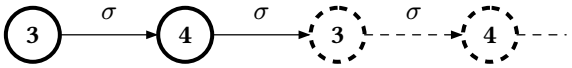
$$\sigma = \{0, 1, 2, 3, 4, 5\} \longmapsto \{2, 0, 1, 4, 3, 5\}$$

This permutation maps **0 to 2, 1 to 0, 2 to 1, 3 to 4, 4 to 3, and 5 to 5.**

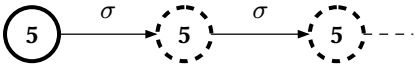
To find the orbits, let's try repeatedly applying σ to each element of the set, starting at 0.



Notice how 3, 4, and 5 are completely unreachable, as we've encountered a repeating cycle. Let's try starting from 3 instead.



We've encountered another repeating cycle! That leaves just 5 left to check.



As we can see, there are three distinct orbits in σ. **They are $\{0, 1, 2\}$, $\{3, 4\}$, and $\{5\}$.**

Write a function that takes in an **int[] of length N containing the integers $[0, N)$** representing a permutation, and returns the number of orbits in the permutation.

Example: σ would be represented in Java by the array `[2, 0, 1, 4, 3, 5]`

```
public static int countOrbits(int[] permutation) {

  _____ count = _____;

  _____<Integer> seen = _____;


  for(int i = 0, i < _____; i++) {

    int currValue = _____;

    if (!_____.contains(currValue)) {

      while(!_____.contains(currValue)) {

        _____;

        currValue = _____;
      }

      _____
    }
  }

  return count;
}
```