CS 61B Spring 2026

Programming Review

Discussion 01: January 20, 2025

In 61B, we'll be learning the Java programming language, with an emphasis on the usage and creation of data structures. Let's kick off the class by reviewing how lists and maps (called dictionaries in Python) work. Below, write the requested Python functions. Use loops. **Do not use any list or dictionary comprehensions (we won't learn them in 61B).**

(a)
```
def evens(list_of_ints: list[int]) -> list[int]:
    """Returns a copy of the list but keeping only the even numbers."""
    return_list = []

    for x in list_of_ints:
      if x % 2 == 0:
        return_list.append(x)

    return return_list
```

(b)
```
<<<<<<< HEAD
  def count_words(list_of_words):
     """Returns a map from each word to its count."""
     counts = {}
=======
  def count_words(list_of_words: list[str]) -> dict[str, int]:
     """Returns a dictionary mapping each word to its count"""
     counts = {}
>>>>>>> refs/remotes/origin/main

    for word in list_of_words:
       if word not in counts:
          counts[word] = 1
       else:
          counts[word] += 1

     return counts
```

(c) Below, we see the Java solution to these problems. Discuss with your group what interesting features you observe in the Java code. If you have any Java veterans in your group, grill them about the weirdness, or of course feel free to ask your TA.

```
public static List<Integer> evens(List<Integer> L) {
    List<Integer> list = new ArrayList<>();
    for (int num : L) {
        if (num % 2 == 0) {
            list.add(num);
        }
    }
    return list;
}
```

```
public static Map<String, Integer> countWords(List<String> words) {
    Map<String, Integer> map = new TreeMap<>();
    for (String word : words) {
        if (map.containsKey(word)) {
            map.put(word, map.get(word) + 1);
        } else {
            map.put(word, 1);
        }
    }
    return map;
}
```

(a) Create a **Dog** class in python. A dog should have two properties: a name and a size. The dog class should have a method called **grow** that increases the dog's size by 1. If the user prints out a dog, it should print the name followed by "the size ", followed by the size, followed by the word " dog" For example, the code below should print "maya the size 1000 dog".

```
dogs = [Dog("maya", 1000), Dog("yipster", 5), Dog("scott", 25)]
print(dogs[0])
```

```
class Dog():

  def __init__(self, name, size):
    self.name = name
    self.size = size

  def grow(self):
    self.size += 1

  def __str__(self):
    return(f"{self.name} the size {self.size} dog")
```

(b) Below, we show the Java solution. As before, discuss with your group what you observe about the code.

```
import java.util.List;

public class Dog  {
    public String name;
    public int size;

    public Dog(String n, int s) {
      name = n;
      size = s;
    }

    public void grow() {
        size += 1;
    }

    @Override
    public String toString() {
        return name + " the size " + size + " dog";
    }

    public static void main(String[] args) {
        List<Dog> dogs = List.of(new Dog("maya", 1000),
                                 new Dog("yipster", 5),
                                 new Dog("scott", 25));
        System.out.println(dogs.get(0));
    }
}
```

(a) Using the example code from earlier in this worksheet, try to write a Java function below which returns the difference between the maximum and minimum item in a `List<Integer>`. You may assume the list has length at least 1. You can get the ith item of a `List` called `L` by calling `L.get(0)`. You can get the size of a list with `L.size()`. There are solutions that don't use either of these functions.

```
public static int maxMinDiff(List<Integer> L) {

  // solution 1:
  return Collections.max(L) - Collections.min(L);

  // solution 2:
  int minVal = L.get(0);
  int maxVal = L.get(0);
  for (int x : L) {
      if (x < minVal) {
          minVal = L;
      }
      if (x > maxVal) {
          maxVal = L;
      }
  }
  return maxVal - minVal;
```

(b) Extra problem: Write a Java function that takes a `List<String>` and returns a map from each String to the list of Strings that immediately follow it (i.e. come right after it). For example, if the input list is `["I", "love", "java", "but", "I", "love", "python", "more"]`, then the output should be:

```
{
  "I": ["love", "love"],
  "love": ["java", "python"],
  "java": ["but"],
  "but": ["I"],
  "python": ["more"]
}
```

```
public static Map<String, List<String>> listOfFollowers(List<String> x) {
```

```
Map<String, List<String>> result = new TreeMap<>();

// We loop up to size() - 1 because the very last word
// has nothing following it.
for (int i = 0; i < x.size() - 1; i++) {
    String currentWord = x.get(i);
    String nextWord = x.get(i + 1);

    if (!result.containsKey(currentWord)) {
        result.put(currentWord, new ArrayList<>());
    }


    result.get(currentWord).add(nextWord);
}

return result;
```