CS 61BL Data Structures & Programming Methodology

Summer 2018

Midterm 2 Solution

This exam has 8 questions worth a total of 25 points and is to be completed in 80 minutes.
The exam is closed book except for two double-sided, handwritten cheat sheets. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.
Write the statement below in the blank provided and sign. You may do this before the exam begins.

"I have neither given nor received any assistance in the taking of this exam."

I have neither given nor received any assistance in the taking of this exam.

Signature: Blear Hug

Question	Points
1	½
2	½
3	5
4	5
5	6
6	4
7	4
8	0
Total	25

Name	Blear Hug
Student ID	1234567890
Lab Section	0 0 1
Name of person to left	Christine Zhou
Name of person to right	Kevin Lin

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are much more complicated than necessary.
- **Work through the problems with which you are comfortable first.** Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful, and **you may not need all lines**. For code-writing questions, **write only one statement per line** and **do not write outside the lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed, but in the event that we do catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not, 'does not compile.'**

1. (½ pt) **Your Thoughts**

Peer pressure is a powerful motivator.

2. (½ pt) **So It Begins** Write the statement on the front page and sign. Write your name, ID, and your lab section. Write the names of your neighbors. Write your name in the corner of every page.

3. (5 pts) **Dynamic Method Selection** Suppose a student is working on their Gitlet project and wants to know what will happen if they define a `Commit` and `MergeCommit` class as shown below.

```
public class Commit {
    public int compareTo(Commit o) { ... }      // A
    public boolean equals(Object o) { ... }     // B
}
public class MergeCommit extends Commit {
    public int compareTo(Commit o) { ... }      // C
    public int compareTo(MergeCommit o) { ... } // D
    public boolean equals(Object o) { ... }     // E
}
```

For each line below, write either the method that will be executed at runtime (**A, B, C, D, E**), or choose **Compile Error** if the line will cause a compilation error, or **Runtime Error** if the line will cause an error at runtime. If an error occurs, assume that the line is removed from the program.

Either write a letter in the blank, or fill in only one circle for each line; do not do both.

```
public class MergeCommitTest {
    public static void main(String[] args) {
        Commit com = new MergeCommit();
        MergeCommit mrg = new MergeCommit();

        com.equals(com);                 E____    ○ Compile Error  ○ Runtime Error

        com.equals(mrg);                 E____    ○ Compile Error  ○ Runtime Error

        mrg.equals(com);                 E____    ○ Compile Error  ○ Runtime Error

        mrg.equals(mrg);                 E____    ○ Compile Error  ○ Runtime Error

        com.compareTo(com);              C____    ○ Compile Error  ○ Runtime Error

        com.compareTo(mrg);              C____    ○ Compile Error  ○ Runtime Error

        mrg.compareTo(com);              C____    ○ Compile Error  ○ Runtime Error

        mrg.compareTo(mrg);              D____    ○ Compile Error  ○ Runtime Error

        ((Commit) com).compareTo(com);   C____    ○ Compile Error  ○ Runtime Error
    }
}
```

Name: Blear Hug

4. **Runtime Analysis** For each problem, give the best and worst-case runtimes in $\Theta(\cdot)$ notation as a function of N. Your answer should be simple with no unnecessary leading constants or summations.

   Recall that the `||` (boolean or) operator *short-circuits* and stops after evaluating the first `true` value.

   Don't spend too much time on these!

   (a) (1 pt) `removeIndex` **Best Case**: $\Theta(1)$ **Worst Case**: $\Theta(N)$

```
public static void removeIndex(int[] arr, int i) {
    // Assume i > 0
    int N = arr.length;
    for (int j = i; j < N; j += 1) {
        arr[j - 1] = arr[j];
    }
}
```

   (b) (2 pts) `recurse` **Best Case**: $\Theta(2^N \log N)$ **Worst Case**: $\Theta(2^N \log N)$

```
public static int recurse(int N) {
    return helper(N, N / 2);
}
private static int helper(int N, int M) {
    if (N <= 1) {
        return N;
    }
    for (int i = 1; i < M; i *= 2) {
        System.out.println(i);
    }
    return helper(N - 1, M) + helper(N - 1, M);
}
```

   (c) (2 pts) `find` **Best Case**: $\Theta(N)$ **Worst Case**: $\Theta(N^2)$

```
public static boolean find(int tgt, int[] arr) {
    int N = arr.length;
    return find(tgt, arr, 0, N);
}
private static boolean find(int tgt, int[] arr, int lo, int hi) {
    if (lo == hi || lo + 1 == hi) {
        return arr[lo] == tgt;
    }
    int mid = (lo + hi) / 2;
    for (int i = 0; i < mid; i += 1) {
        System.out.println(arr[i]);
    }
    return arr[mid] == tgt || find(tgt, arr, lo, mid) || find(tgt, arr, mid, hi);
}
```

5. **Functions & Streams** Java language reference sheet can be found on the last page of the exam.

(a) (2 pts) Implement `odds` which returns a list of only odd integers using a single Java statement, though it may be split over multiple lines. Use `n % 2 == 1` to check if a number, `n`, is odd.

```
List<Integer> odds(List<Integer> values) {

    return values.stream().filter(i -> i % 2 == 1)

                          .collect(Collectors.toList());
}
```

(b) (2 pts) Implement `MapFunction`, a class of type `Function` which takes a `Map<K, V>` and a default value of type `V`. `apply` returns the value associated with the key in the map if the key is in the map, otherwise it returns the `valueIfNotFound`. Make sure to fill in the generic types in the class and method declaration, or leave them blank if they're not necessary.

```
public class MapFunction<K, V> implements Function<K, V> {
    private final Map<K, V> map;
    private final V valueIfNotFound;
    public MapFunction(Map<K, V> m, V v) {
        map = m;
        valueIfNotFound = v;
    }
    public V apply(K key) {

        if (map.containsKey(key)) {

            return map.get(key);

        } else {
            return valueIfNotFound;
        }
    }
}
```

(c) (2 pts) Implement `mapThenReLU`, which first applies a new `MapFunction` (using the map and default value given in the arguments) and then applies the `relu` operator, where $relu(x) =$ `Math.max`$(0, x)$, to each value in the stream. Assume that `MapFunction` is correctly implemented.

```
Stream<Double> mapThenReLU(Stream<Double> data, Map<Double, Double> m, Double v) {

    return data.map(new MapFunction<Double, Double>(m, v))

               .map(x -> Math.max(0, x));
}
```

Name: Blear Hug

6. (4 pts) **Iterators** Implement `CharIterator`, an iterator which takes a `String[]`. Calls to `next` return the next character in each string in order, from the first character of the first string all the way to the last character of the last string. For a `CharIterator({"hi", "ya"})`, calling `next` will first return `'h'`, then `'i'`, then `'y'`, then `'a'`. The behavior for subsequent calls to `next` is undefined.

```
public class CharIterator implements Iterator<Character> {
    String[] strings; int sIndex; char[] word; int wIndex;
    public CharIterator(String[] input) {
        strings = input;
        if (strings != null && strings.length > 0) {
            word = strings[0].toCharArray();
            sIndex = wIndex = 0;
        } else {
            word = null;
        }
    }
    public boolean hasNext() {

        return word != null && (wIndex < word.length || sIndex < strings.length - 1)

            ;
    }
    public Character next() {

        if (wIndex < word.length) {

            Character toReturn = word[wIndex];

            wIndex += 1;

            return toReturn;
        } else {

            sIndex += 1;

            wIndex = 0;

            word = strings[sIndex].toCharArray();

            return next();
        }
    }
}
```

7. **Disjoint Sets** In this problem, we will analyze a simplified implementation of path compression called *path halving*. In the `find` method for a **weighted quick-union with *path halving***, instead of making every node on the path point to the root, we halve the path length by making every other node in the path point to its grandparent. **The `parent` function will always return the parent or the root, never the size.** Consider the `find` implementation below.

```
public int find(int p) {
    if (p < 0 || p >= data.length) {
        throw new IllegalArgumentException("Invalid vertex: " + p);
    }
    while (p != parent(p)) {
        data[p] = parent(parent(p));    // path halving
        p = parent(p);
    }
    return p;
}
```

(a) (1 pt) Give the best and worst-case runtimes for `find` in $\Theta(\cdot)$ notation as a function of N, the total size of the disjoint sets data structure.

Best Case: $\Theta(1)$ **Worst Case**: $\Theta(\log N)$

(b) (1 pt) Consider a **weighted quick-union with *path quartering***. Suppose we replace the line marked, `path halving`, with `data[p] = parent(parent(parent(parent(p))))`. Give the best and worst-case runtimes for `find` in $\Theta(\cdot)$ notation as a function of N.

Best Case: $\Theta(1)$ **Worst Case**: $\Theta(\log N)$

(c) (1 pt) A *fully-connected* disjoint sets object is one in which `connected` returns true for any arguments, due to prior calls to `union`. The height is the number of links from the root to the deepest leaf, so a tree with 1 element has a height of 0. Give the least and greatest-possible height for a *fully-connected* **weighted quick union with *path halving*** with **9 items**.

Give an exact value; do not use asymptotic notation.

Least Height: 1 **Greatest Height**: 3

(d) (1 pt) Give the best and worst-case height for a *fully-connected* **weighted quick union with *path quartering*** with **15 items**.

Give an exact value; do not use asymptotic notation.

Least Height: 1 **Greatest Height**: 3

8. (0 pts) **PNH** Which *daimyō* is believed to have been one of the first Japanese to eat ramen?

Tokugawa Mitsukuni

Abstract Interface Reference

```java
public interface Map<K, V> {
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
}
```

Function Reference

```java
public interface Function<T, R> {
    R apply(T t);
}
public interface Predicate<T> {
    boolean test(T t);
}
public interface BinaryOperator<T> extends BiFunction<T, T, T> { ... }
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
public interface Consumer<T> {
    void accept(T t);
}
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Stream Reference

```java
public interface Stream<T> {
    Stream<T> filter(Predicate<? super T> predicate);
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
    Stream<T> sorted(Comparator<? super T> comparator);
    void forEach(Consumer<? super T> action);
    Optional<T> reduce(BinaryOperator<T> accumulator);
    <R, A> R collect(Collector<? super T, A, R> collector);
}
public class Collectors {
    static <T> Collector<T, ?, List<T>> toList() { ... }
}
public class Optional<T> {
    T get() { ... }
    T orElse(T other) { ... }
}
```