

UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #1, Spring 2018

This test has 8 questions worth a total of 160 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out 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.”

Signature: _____

| # | Points | # | Points |
|--------------|--------|---|------------|
| 0 | 0.75 | 6 | 20 |
| 1 | 19 | 7 | 23 |
| 2 | 12 | 8 | |
| 3 | 16.25 | | |
| 4 | 37 | | |
| 5 | 32 | | |
| TOTAL | | | 160 |

Name: _____

SID: _____

Three-letter Login ID: _____

Login of Person to Left: _____

Login of Person to Right: _____

Exam Room: _____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones 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.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to 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.’**
- indicates that only one circle should be filled in.
- indicates that more than one box may be filled in.
- For answers which involve filling in a or , please fill in the shape completely.

Optional. Mark along the line to show your feelings
on the spectrum between ☹ and ☺.

Before exam: [☹ _____ ☺].
After exam: [☹ _____ ☺].

0. So it begins (0.75 points). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. Enjoy your free 0.75 points ☺.

1. Static Dada.

a) (10 points) Consider the class shown below. Next to the lines with blanks, write the result of the print statement. **No syntax errors or runtime errors occur.**

```
public class Dada {
    private static String[] rs;

    /** Prints out the given array, i.e. if d contains two Strings
     * with names "alice" and "bob", this method will print "alice bob ".
     */
    private static void printStringArray(String[] s) {
        for (int i = 0; i < s.length; i += 1) {
            System.out.print(s[i] + " "); }
        System.out.println();
    }

    public static void main(String[] args) {
        String a = "alice";
        String b = "bob";
        String c = "carol";
        String d = "dan";
        String[][] twod = {{a, b}, {c, d}};
        String[] X = twod[1];
        printStringArray(X); // _____
        Dada.rs = X;
        String[] Y = Dada.rs;
        Y = new String[]{d, c};
        Dada.rs[1] = "eve";
        printStringArray(Dada.rs); // _____
        printStringArray(Y); // _____
        printStringArray(twod[0]); // _____
        printStringArray(twod[1]); // _____
    }
}
```

Login: _____

b) (9 points) Suppose we add new methods to Dada called `fillOne` and `fillMany` and replace `main` as shown below. Fill in the print statements. The Dada class is otherwise unchanged.

```
private static void fillMany(String[] d) {
    System.arraycopy(Dada.rs, 0, d, 0, d.length);
}

private static void fillOne(String d) { d = Dada.rs[0]; }

public static void main(String[] args) {
    String a = "alice";
    String b = "bob";
    String c = "carol";
    String d = "dan";
    String[][] twod = {{a, b}, {c, d}};

    Dada.rs = new String[]{"fritz", "gritz"};
    String[] X = twod[0];
    printStringArray(X);           // _____
    fillOne(X[0]);
    printStringArray(X);           // _____
    fillMany(X);
    printStringArray(X);           // _____
}
```

2. What It Do (12 Points).

a) (8 points). Consider the code below.

```
public static int f(int x) {  
    if (x == 1) {  
        return 1;  
    }  
    return 2 * f(x / 2);  
}
```

Describe as **succinctly** as possible what this method does when executed for **all possible values of x**. If the behavior is different depending on x, describe the behavior in every interesting case. Remember that integer division in Java rounds down, i.e. $3/2$ yields 1.

b) (4 points). Consider the code below.

```
public static void g(IntList x) {  
    if (x == null) { return; }  
    g(x.rest);  
    x.rest = x;  
}
```

Draw a box and pointer diagram that shows the result of executing the following two lines of code. If any objects are not referenced by anything else (i.e. are garbage collected), you may omit drawing them if you prefer. If you need it, the `IntList` definition is on page 7. If `g` never finishes because it gets stuck in an infinite loop, write “Infinite Loop” instead of drawing a diagram.

```
IntList L = IntList.of(3, 4, 5); //creates an IntList containing 3, 4, and 5  
g(L);
```

Login: _____

3. **KeyGate (16.25 points)**. Suppose we have the classes defined below, with 3 lines marked with **UK**, **USK**, and **UF**.

```
public class Fingerprint {...}
public class Key { ... }
public class SkeletonKey extends Key { ... }

public class StandardBox { public void unlock(Key k) { ... } } // UK

public class BioBox extends StandardBox {
    public void unlock(SkeletonKey sk) { ... } // USK
    public void unlock(Fingerprint f) { ... } // UF
}
```

For each line below, fill in exactly one bubble. **If a line causes an error, assume it is commented out before the following lines are run.**

```
public static void doStuff(Key k, SkeletonKey sk, Fingerprint f) {
    StandardBox sb = new StandardBox();
    StandardBox sbbb = new BioBox();
    BioBox bb = new BioBox();
```

| | Compile Error | Runtime Error | UK Runs | USK Runs | UF Runs |
|----------------------------------|-----------------------|-----------------------|---------------------------|-----------------------|-----------------------|
| sb.unlock(k); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| sbbb.unlock(k); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| bb.unlock(k); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| sb.unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| sbbb.unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| bb.unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| sb.unlock(f); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| sbbb.unlock(f); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| bb.unlock(f); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| bb = (BioBox) sbbb; | <input type="radio"/> | <input type="radio"/> | ← Leave blank if no error | | |
| ((StandardBox) bb).unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| ((StandardBox) sbbb).unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| ((BioBox) sb).unlock(sk); | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |

}

4. **Sans.** Implement the methods below. For reference, the `IntList` class is defined at the bottom of the next page.

a) (7 points). `/** Non-destructively creates a copy of x that contains no y. */`
public static int[] sans(int[] x, int y) {
 int[] xclean = new int[x.length];
 int c = 0;
 for (int i = 0; i < x.length; i += 1) {
 if (_____) {

 }
 }
 int[] r = _____;
 System.arraycopy(_____, _____);
 return r; // arraycopy parameters are:
 // srcArr, srcStartIdx, destArr, destStartIdx, numToCopy
 // where src->source, dest->destination, Idx->index
 }

b) (9 points). `/** Non-destructively creates a copy of x that contains no y. */`
public static IntList ilsans(IntList x, int y) {
 if (_____) {
 return _____
 }
 if (_____) {
 return _____
 }
 return new _____
 }

c) (9 points). `/** Destructively creates a copy of x that contains no y. You may not use new. */`
public static IntList dilsans(IntList x, int y) {
 if (_____) {

 }

 if (x.first == y) {
 return _____
 }
 return _____
 }

Login: _____

d) (12 points). Suppose we want to write tests for `sans` and `ilsans`. Fill in the code below with a JUnit test to see if each method behaves as expected for one example input. Do not write a test for null inputs. Reminder that `IntList.of(4, 5, 6)` creates an `IntList` containing the values 4, 5, and 6. Assume the methods on the previous page are all part of a class called `Sans`, i.e. they are invoked as `Sans.sans`.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestSans {
    @Test
    public void testSans() { // TEST THE ARRAY VERSION OF SANS
        int[] x = _____
        int y = _____
        int[] expected = _____
        int[] actual = _____
        _____
        _____
    }

    @Test // TEST THE NON-DESTRUCTIVE INTLIST VERSION OF SANS
    public void testIlsans() {
        IntList x = IntList.of(_____
        int y = _____
        IntList expected = IntList.of(_____
        IntList actual = _____
        _____
        _____
        _____
    }
}
```

For reference, part of the `IntList` class definition is given below:

```
public class IntList {
    public int first;
    public IntList rest;
    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
    public IntList() {}
    public static IntList of(Integer... args) { /* works correctly */ }
    public boolean equals(Object x) { /*works correctly with assertEquals*/ }
    ...
}
```

5. A Needle in ArrayStack. The Stack interface is given below. A Stack is basically like the proj1 Deque, where `push` is like “addLast”, and `pop` is like “removeLast”. For example, if we call `push(5)`, `push(10)`, `push(15)`, then call `pop()`, we’d get 15. If we call `pop()` again, we get 10.

```
public interface Stack<Item> {
    void push(Item x); // places an item on “top” of the stack
    Item pop();        // removes and returns “top” item of the stack
    int size();        // returns the number of items on the stack
}
```

a) (14 points). Fill in the ArrayStack implementation below. To ensure efficient memory usage, double the array size when full, halve the array size when $< 1/4$ full, and avoid storing unnecessary references. The if conditions for resizing during `push` and `pop` are provided for you in the skeleton code.

```
public class ArrayStack<Item> implements Stack<Item> {
    private Item[] items;

    _____

    public ArrayStack() { // initial array size is 8
        items = (Item[]) new Object[8];
        _____
    }

    private void resize(int capacity) { // resizes array to given capacity
        _____
        _____
        _____
        _____
    }

    public void push(Item x) {
        if (usageRatio() == 1) { _____ }
        _____
        _____
    }

    public Item pop() { // returns null if stack is empty
        if (_____) { return null; }
        if (usageRatio() < 0.25 && items.length > 8) { _____ }
        _____
        _____
        _____
        _____
    }

    public int size() { return _____ }
    private double usageRatio() { return ((double) size()) / items.length; }
}
```


6. **Combine.** The `Combine.combine` method takes a `ComFunc` and an integer array `x` and uses the `ComFunc` to “combine” all the items in `x`. For example, if we have an implementation of `ComFunc` called `Add` that adds two integers, and we call `combine` using the `Add` class on the array `{1, 2, 3, 4}`, the result will be 10, since $1 + 2 + 3 + 4$ is 10.

a) (16 points). Fill in the `combine` method below. If the array is of length 0, the result should be 0, and if the array is of length 1, the result should be the number in the array. For full credit use recursion. For 75% credit, you may use iteration. **You may create a private helper function in the space provided.**

```
public interface ComFunc {
    int apply(int a, int b); // apply(a, b) must equal apply(b, a)
}
public class Combine {
    public static int combine(ComFunc f, int[] x) {
        if (_____) {
            return _____;
        }
        if (_____) {
            return _____;
        }
        _____
        _____
    }
    // your private helper function cannot create new arrays (too slow)
    private static _____(_____) {
        _____
        _____
        _____
        _____
    }
}
```

b) (4 points). Suppose we have a method that adds two numbers, as shown below.

```
public class Add implements ComFunc {
    public int apply(int a, int b) {
        return a + b;
    }
}
```

Fill in the method below so that it prints out the correct result. You may use your answer from part a. Even if you left part a blank or think it be incorrect, you can assume that everything works as expected.

```
public static int sumAll(int[] x) { // sumAll is not a member of Combine
    return _____
}
```

Login: _____

7. The Downside of Default. Consider the `ListOfInts` interface below. `addLast`, `get`, and `size` behave exactly as your `Deque` interface from project 1A. `set(int i, int value)` sets the *i*th integer in the list equal to `value`. `plusEquals` adds each `int` in `x` to the corresponding `int` in the current list, i.e. if we call have a list `L = [2, 3, 4]` and we call `L.plusEquals([5, 6, 7])`, then after the call is complete, `L` will contain the values `[7, 9, 11]`. **If the lists are not of the same length, `plusEquals` should have no effect.**

a) (6 points). Fill in the `plusEquals` method below.

```
public interface ListOfInts {
    public void addLast(int i);
    public int get(int i);
    public int size();
    public void set(int i, int value);
    default public void plusEquals(ListOfInts x) { // assume x is non-null
        if (_____){ return; }
        for (int i = 0; _____) {
            this.set(i, _____);
        }
    }
}
```

b) (10 points). The `DLListOfInts` class is an implementation of `ListOfInts` that stores a doubly linked list of integers, similar to your `LinkedListDeque` class. For a `DLListOfInts`, the default `plusEquals` method will be very slow, since it relies on `get` and `set`. Fill in the `plusEquals` method so that it behaves as in part a, but has a more efficient runtime, i.e. doesn't rely on `get` or `set`. You must use iteration. Assume that each list has a single sentinel node that points at itself when the list is empty, just like in lecture and in the recommended approach for proj1a.

```
public class DLListOfInts implements ListOfInts {
    public class IntNode {
        public int item;
        public IntNode next, prev;
    }
    public IntNode sentinel;
    public int size;
    @Override
    public void plusEquals(DLListOfInts x) {
        if (_____){
            _____
        }

        _____
        for (IntNode p = sentinel.next; _____; _____) {
            _____
            _____
        }
    }
} ...
```

c) (7 points) The method `sumOfLists` given below is supposed to take an array of `DLListOfInts` and returns a `DLListOfInts` that is equal to the element-wise sum of all of the lists. For example if the array contains three lists that hold `[2, 2, 2]`, `[1, 2, 3]`, and `[3, 3, 3]`, respectively, the method should return a `DLListOfInts` that contains `[6, 7, 8]`. The method should be non-destructive.

```
public class PartC {
    /** Non-destructively computes the sum of the given lists. Assumes
     * that all lists are of the same length and that none are null. */
    public static DLListOfInts sumOfLists(DLListOfInts[] lists) {
        ListOfInts result = lists[0];

        for (int i = 1; i < lists.length; i += 1) {
            result.plusEquals(lists[i]);
        }
        return result;
    }
}
```

What mistakes (if any) are there in `sumOfLists`? Note: The fact that the method makes the listed assumptions (“all lists are of the same length and none are null”) is not a mistake, it’s an assumption.

8. PNH (0 points). What two catastrophic events are believed to be responsible for the creation of almost all of the gold on the earth?