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."

Meow meow meow meow meow meow meow meow meow purr.

Signature: Junhao Wang

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

Name: Junhao Wang

SID: I don't know~

Three-letter Login ID: Cat

Login of Person to Left: Dog

Login of Person to Right: Snake

Exam Room: Restroom

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 **<u>you may not need all lines</u>.**
- 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. **<u>Unless we specifically give you the option, the correct answer is not 'does not compile.'</u>**
- ○ 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: [☹_v_______________☺].
After exam: [☹_______________v☺].

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);                 //____carol dan____
        Dada.rs = X;
        String[] Y = Dada.rs;
        Y = new String[]{d, c};
        Dada.rs[1] = "eve";
        printStringArray(Dada.rs);           //____carol eve____
        printStringArray(Y);                 //____dan carol____
        printStringArray(twod[0]);           //____alice bob____
        printStringArray(twod[1]);           //____carol eve____
    }
```

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);          //   alice bob
    fillOne(X[0]);
    printStringArray(X);          //    fritz bob
    fillMany(X);
    printStringArray(X);          //    fritz gritz
}
```

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 **<u>all possible values of x</u>.** 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.

f(0): infinite loop
f(1): return 1
f(2): 2 * f(1) -> 2 * 1 -> 2
f(3): 2 * f(1) -> 2 * 1 -> 2
f(4): 2 * f(2) -> 2 * 2 * f(1) -> 2 * 2 * 1 -> 4
f(5): 2 * f(2) -> 2 * 2 * f(1) -> 2 * 2 * 1 -> 4
f(6): 2 * f(3) -> 2 * 2 * f(1) -> 2 * 2 * 1 -> 4
f(7): 2 * f(3) -> 2 * 2 * f(1) -> 2 * 2 * 1 -> 4
f(8): 2 * f(4) -> 8

f(-1): 2 * f(-1/2) -> 2 * f(0) -> infinite loop
f(-2): 2 * f(-2/2) -> infinite loop

when x <= 0: infinite loop
when x > 0,
  if n == 2^N: f(n) will return: n
  otherwise: f(n) will return: 2^N, (N <= n < N + 1)

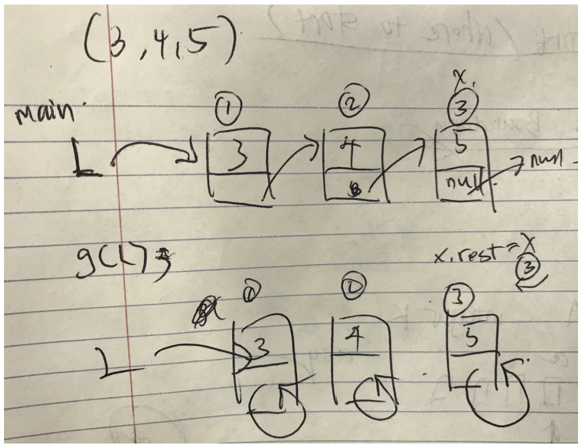
prints out the largest power of 2 that is smaller than x

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);
```



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
}
```

It's important to know which one is overloading or overriding.

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();
```

static type in compile time

	Compile Error	Runtime Error	UK Runs	USK Runs	UF Runs
`sb.unlock(k);`	○	○	✓	○	○
`sbbb.unlock(k);`	○	✓	✓	○	○
`bb.unlock(k);`	✓	○	✓	○	○
`sb.unlock(sk);`	○	○	✓	○	○
`sbbb.unlock(sk);`	○	○	✓	✓	○
`bb.unlock(sk);`	○	○	○	✓	○
`sb.unlock(f);`	✓	○	○	○	○
`sbbb.unlock(f);`	✓	○	○	○	○
`bb.unlock(f);`	○	○	○	○	✓
`bb = (BioBox) sbbb;`	○	○	← Leave blank if no error		
`((StandardBox) bb).unlock(sk);`	○	○	✓	✓	○
`((StandardBox) sbbb).unlock(sk);`	○	○	✓	✓	○
`((BioBox) sb).unlock(sk);`	○	✓	○	○	○

```
}
```

No override here!

Match first, then consider if selection may occur.

not specific, but compatible

Since there is no override, dynamic method selection won't occur.

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 (____x[i] != y____) {
                ____xclean[c] = x[i];____
                ____c += 1;____
            }
        }
        int[] r = ____new int[c];____
        System.arraycopy(____xclean, 0, r, 0, c____);
        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 (____x == null____) {
            return ____null;____
        }
        if (____x.first == y____) {
            return ____ilsans(x.rest, y);____
        }
        return new ____IntList(x, ilsans(x.rest, y));____
}
```

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 (____x == null____) {
            ____return null;____
        }
        ____x.rest = dilsans(x.rest, y);____
        if (x.first == y) {
            return ____x.rest;____
        }
        return ____x____
}
```

d) (12 points). Suppose we want to write tests for and `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 = new int[3] {4, 5, 6};
      int y = 5;
      int[] expected = new int[2] {4, 6};
      int[] actual = Sans.sans(x, y);
      assertArrayEquals(expected, actual);
      assertArrayEquals(new int[3] {4, 5, 6}, x);
  }

  @Test // TEST THE NON-DESTRUCTIVE INTLIST VERSION OF SANS
  public void testIlsans() {
      IntList x = IntList.of(4, 5, 6);
      int y = 5
      IntList expected = IntList.of(4, 6);
      IntList actual = Sans.ilsans(x, y);
      assertEquals(expected, actual);
      assertEquals(IntList.of(4, 5, 6), x);
      ______________________________________
  }
}
```

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;
     private int size;

    public ArrayStack() { // initial array size is 8
        items = (Item[]) new Object[8];
           size = 0;
    }
    private void resize(int capacity) { // resizes array to given capacity
                Item[] newItems = (Item[]) new Object[capacity];
                for (int i = 0; i < size; i += 1) {
                    newItems[i] = items[i];        or: System.arraycopy(items, 0, newItems, 0, size);
                 }
                                items = newItems;
    }
    public void push(Item x) {
        if (usageRatio() == 1) {           resize(size * 2);              }
                items[size] = x;
                size += 1;
    }
    public Item pop() { // returns null if stack is empty
        if (     size == 0                        ) { return null; }
        if (usageRatio() < 0.25 && items.length > 8) {  resize(size / 2);     }
             size -= 1;                                     bug, it should be
             Item ret = items[size];                        resize(items.length / 2);
             items[size] = null;
             return ret;

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

b) **(18 points)** Suppose we want to add a **default method** `purge(Item x)` **to the `Stack` interface** that eliminates all instances of x from the `Stack`, but leaves the stack otherwise unchanged. When comparing two items, remember to use `.equals` instead of `==`. You may assume the items in the stack are not `null`, and you may assume that `x` is not `null`.

For example, suppose we create a Stack and call `push(1)`, `push(2)`, `push(3)`, `push(2)`, `push(2)`, `push(2)`, then call `purge(2)`, the stack would be reduced to size 2, and would have 3 on top and 1 on the bottom.

You may use an `ArrayStack` for this problem and assume it works correctly, even if you didn't finish part a or are unsure of your answer. **You may <u>not</u> explicitly instantiate any other class or any array of any kind**, e.g. no `new LinkedListDeque<>()`, `new int[]`, etc.

```
public interface Stack<Item> {
    public void push(Item x);
    public Item pop();
    public int size();
    public default void purge(Item x) {
            Stack<Item> temp = new ArrayStack<>();

            while (size() != 0) {
                Item it = pop();
                if (!it.equals(x)) {
                    temp.push(it);
                }
            }

            while (temp.size() != 0) {
                Item it = temp.pop();
                push(it);
            }
    }
}
```

Recursive Version:

```
if (size() == 0) {
    return 0;
}

Item top = pop();
purge(x);
if (!x.equals(top)) {
    push(top);
}
```

Midterm 1 Leisure Region. Please relax and have a nice time in this region.

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. **<u>You may create a private helper function in the space provided.</u>**

```
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 (__x == null || x.length == 0__){
        return ____0_______;
    }
    if (__x.length == 1___){
        return _____x[0]____;
    }
    __________int sum = comb(f, x, 0);__________
    __________return sum;_______________________
  }
  // your private helper function cannot create new arrays (too slow)
  private static __int__ __comb__ (_____ComFunc f, int[] x, int index_____) {
    ________if (index == x.length - 1) {___________
    _____________return x[index];__________________
    ________}______________________________________
    ________int ret = f.apply(x[index], comb(f, x, index + 1));
    ________return ret;____________________________
  }
}
```

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 _______Combine.combine(new Add(), x);___________
}
```

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 ith 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 (__size() != x.size()__){ return; }
        for (int i = 0; __i < size(); i += 1__) {
            this.set(i, __get(i) + x.get(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 (__size != x.size || x == null__) {
            __return;__
        }

        __IntNode ptr = x.sentinel.next;__
        for (IntNode p = sentinel.next; __p != sentinel__; __p = p.next__) {
            __p.item += ptr.item;__
            __ptr = ptr.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.

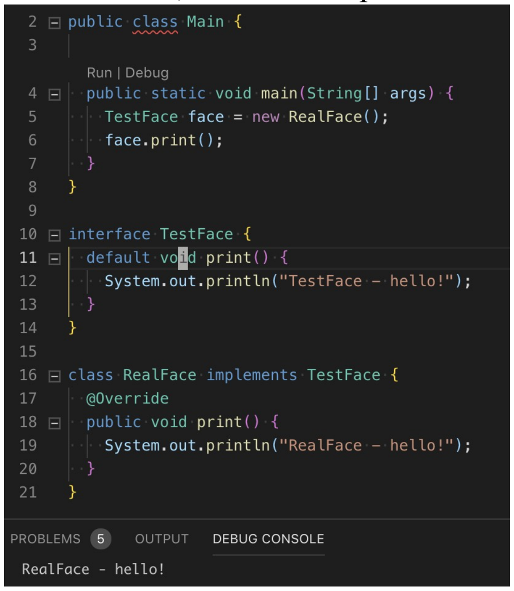
```
public class PartC {
     /** Non-destructively computes the sum of the given lists. Assumes
       * that all lists are of the same length and that none are null. */
     public static DLListOfInts sumOfLists(DLListOfInts[] lists) {
         ListOfInts result = lists[0];

         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.

It is destructive!

- Return type is upcast from result, so the code will not compile.
- A 0-length array input will cause an ArrayIndexOutOfBoundsException.

??? why this one???
- result.plusEquals calls the default plusEquals (in ListOfInts), which is much slower.



```
public class Main {

    Run | Debug
    public static void main(String[] args) {
        TestFace face = new RealFace();
        face.print();
    }
}

interface TestFace {
    default void print() {
        System.out.println("TestFace - hello!");
    }
}

class RealFace implements TestFace {
    @Override
    public void print() {
        System.out.println("RealFace - hello!");
    }
}
```

```
RealFace - hello!
```

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?

- Massive meteorite shower
- Collision between neutron stars