CMSC 206: Data Structures
Harry Potter and the Practice Final Exam

We last left our remaining two heroes, Harry and Hermione, entering a dark chamber. What they hear is surely unexpected:

Surprised you are to find me here,
a room so small and dark,
For I'm the Hogwarts Sorting Hat;
I loathe a room so stark.
Yet I've been told to spend my time
Beneath the lowest floor,
Watching close and looking out
While guarding the next door.
So if you two should wish to pass
And seek what I protect,
Then you must answer questions five
And get them all correct.
These questions will have you recall
A recent lab report,
For my chief purpose in this life
Is knowing how to sort.
Enough prelude, I've gone too long,
It's time that we've begun,
So get yourself all good and primed
For question number one.

1. **(4 pts)** *You have a list all out of order;*
You wish to make it right.
Of all the sorts that you have learned
Selection sort you write.
For these six numbers, given here,
4 6 1 5 2 3
Perform three swaps, and tell me back
The numbers you now see:

A. 1 2 3 4 6 5

B. 1 2 3 4 5 6

C. 4 1 5 2 6 3

D. 1 2 3 5 6 4

E. 1 4 6 2 5 3

2. **(4 pts)** *Good job right there; you did that well.*
Now here's a harder quest.
If I describe a list to you,
Tell me which sort is best.
This list I have goes small to big,
With every number placed
Exactly where the number fits.
Now there's no time to waste.
Which sort of those ones given here
We know will finish first
When processing a sorted list?
(Beware! Don't give the worst!)

A. Selection Sort

B. Insertion Sort

C. Quick Sort

D. Merge Sort

E. Heap Sort *(not covered in CS206)*

3. **(4 pts)** *Quick sort has a running time*
That is apt to change.
To know its time for sure, we need
The numbers to arrange.
Among the lists we have below,
Go choose and tell to me,
The one that quick sort handles best,
The fastest one you see.
`partition` *is a vital part*
of quick sort that we penned;
Consider what you wrote in class,
With the pivot at the end.
(In CS206, we put the pivot at the beginning.)

A. 1 2 3 4 5 6 7 8

B. 8 7 6 5 4 3 2 1

C. 1 8 2 7 3 6 4 5

D. 4 1 2 3 8 5 6 7

E. 4 5 3 6 2 7 1 8

4. **(4 pts)** *So many sorts of sorts exist,*
It's hard to keep them straight;
Choose the comment writ below
To which you best relate.
I never want to be obscure
Or not get through to you,
So to repeat the thought I have,
Select one that is true.

A. Heap sort works by creating a separate max-heap, storing the elements there, and then inserting successive maximum elements in the original array, working backwards. *(Hint to CS206 students: it's not this one.)*

B. Insertion sort works by finding the minimum element in a list and then inserting that element at its proper location.

C. Merge sort requires extra space to work. It is neither the slowest nor the fastest algorithm we have learned.

D. Bubble sort works by swapping neighboring elements until the list is sorted. It is the fastest of the $O(n^2)$ sorts.

E. In the best case, selection sort can run in $O(n)$ time.

5. **(4 pts)** *One question left, and then you're done;*
You'll pass beyond this wall.
So here we are then, question five,
The hardest of them all.
So, read my code and study close;
An error you will find.
You tell me what that error is,
And you might save mankind.

```
/* Merges two sorted lists
 * preconditions:  start < mid < end;
 *                 the first sorted list ranges
 *                 from start to (mid - 1);
 *                 the second sorted list ranges
 *                 from mid to (end - 1)
 * postconditions: the list from start to (end - 1)
 *                 is sorted.  This method runs
 *                 in O(n) time, where n is the
 *                 length of the list.
 */
public void merge(int[] arr, int start, int mid, int end)
{
       int s = start, m = mid, i = 0;
       int[] temp = new int[end - start];
       while(i < temp.length)
       {
              if(arr[s] < arr[m])
              {
                     temp[i] = arr[s];
                     i++;
                     s++;
              }
              else
              {
                     temp[i] = arr[m];
                     i++;
                     m++;
              }
       }
       while(s < mid)
       {
              temp[i] = arr[s];
              i++;
              s++;
       }
       while(m < end)
       {
              temp[i] = arr[m];
              i++;
              m++;
       }
       while(i >= 0)
       {
              arr[start + i] = temp[i];
              i--;
       }
}
```

A. This method will sometimes cause an `ArrayIndexOutOfBoundsException`.

B. This method will sometimes cause a `NullPointerException`.

C. This method will sometimes fail to merge the lists correctly; it will never throw an exception.

D. This method merges the lists in reverse order; the resulting list goes from greatest to least.

E. The use of this method in a recursive `mergeSort` method will lead to an `OutOfMemoryError` because it allocates memory.

Computer wizards you both are,
Showing that you're smart.
And now, as promised, I step aside
And watch you two depart.

After proving themselves to the Sorting Hat, Harry and Hermione advanced to the next room. There, they see 7 bottles of different size containing a variety of potions. No sooner do they cross the threshold to the room than a fire springs up behind them and before them, trapping the two. The fire in the doorway leading back is purple; the fire in the doorway leading onward is black. As experienced wizards, Harry and Hermione instantly knew that one of the potions would allow them to proceed, and one would allow them to go back. But which ones?

Next to the potions was a piece of paper, giving facts about each. However, the facts were not forthcoming – each just gave a little information. For example, one fact is that neither potion at the ends allows you to proceed forward.[1]

What we must do for this problem is to track a set of facts that we know, using those facts to derive new ones. We must also track possibilities for each potion. As we gather facts, we will be able to eliminate possibilities. When only one possibility remains, we know what that potion holds!

Here are the relevant classes:

```
public class Potion
{
     public int hashCode()
     { /* implementation not shown */ }

     /* other methods & fields not shown */
}

// this class is used to represent a known fact
public class Fact
{ /* implementation not shown */ }
```

[1] See the end of this exam for the full set of facts and an extra-credit problem based on these facts.

```
/* this class is used to represent a possibility.  For
 * example, an object of this class might denote the
 * possibility that a potion is the one that allows us to
 * move forward.  A different object of this class might
 * denote the possibility that a potion is instant poison.
 */
public class Possibility
{
     /* returns true if this represents that a potion might
      * allow the drinker to proceed through the black
      * fire.
      */
      public boolean isProceed()
      { /* implementation not shown */ }

      /* other members & fields not shown */
}

public class PotionProblem
{
     // the potions we're dealing with

     private List<Potion> potions;

     // our current set of known facts
     private Set<Fact> facts;

     /* a map from potions to sets of possibilities.  This
      * tracks what different potions are possible for a
      * given potion object.
      */
     private Map<Potion, Set<Possibility>> potionPosse;

     /* Creates a PotionProblem.
      * postconditions: All fields are properly set up.
      *                 potionPosse is initialized so that
      *                 each possibility set contains every
      *                 possibility.
      *                 facts is an empty set of Facts.
      */
     public PotionProblem(/* parameters not shown */)
     { /* implementation not shown */ }
```

```
/* Derives new facts from old ones.
 * preconditions:  The givens are a set of facts.
 * postcontitions: A new set of facts is returned.
 *                 This new set of facts may be larger
 *                 than the givens if new facts were
 *                 derived.  The new set cannot be
 *                 smaller than the givens.
 */
private Set<Fact> deriveNewFacts(Set<Fact> givens)
{ /* implementation not shown */ }

/* Derives all possible new facts from old ones.
 * preconditions:  The facts field is set up.
 * postconditions: The facts field holds a set of all
 *                 facts that can possibly be derived.
 */
private void deriveAll()
{ /* to be implemented below */ }

/* Gets a set of Possibilities that do not apply for
 * the potion at the index given.
 * preconditions:  The facts field is set up;
 *                 0 <= index < potions.size()
 * postconditions: No field is changed;
 *                 Returns a set of Possibilities that
 *                 are not possible for the given
 *                 potion.
 */
private Set<Possibility> getImpossibilities(int index)
{ /* implementation not shown */ }

/* Updates the possibility sets for a potion.
 * preconditions:  The facts field is set up;
 *                 0 <= index < potions.size()
 * postconditions: The potionsPosse field is updated
 *                 so that it does not have any
 *                 impossibilities for the potion at
 *                 the given index.  It is impossible
 *                 for this method to increase the
 *                 size of a possibility set.
 */
private void updatePosse(int index)
{ /* to be implemented below */ }
```

```
    /* Updates all the possibility sets.
     * postconditions: The possibility set is updated for
     *                 every potion.
     */
    private void updateAllPosse()
    { /* to be implemented below */ }

    /* Uses a Queue of facts to eliminate possibilities,
     * in the hopes of finding the potion that moves us
     * forward.
     * preconditions:  queue is a valid queue of facts;
     *                 all fields are initialized properly
     * postconditions: returns the Potion that will allow
     *                 us to proceed if we can identify
     *                 the potion uniquely from the given
     *                 facts; otherwise, returns null.
     */
    public Potion findGoodPotion(Queue<Fact> queue)
    { /* to be implemented below */ }
}
```

6. **(8 pts)** Write the method `deriveAll`. This method takes the facts stored in the `facts` field and repeatedly calls `deriveNewFacts` until the set of facts known is no longer changing. The idea behind this method is that sometimes deriving one new fact can immediately lead us to other new facts, but we need to call `deriveNewFacts` multiple times to make sure we've exhausted our logic capabilities. The final set of facts should be stored back in the `facts` field.

```
/* Derives all possible new facts from old ones.
 * preconditions:  The facts field is set up.
 * postconditions: The facts field holds a set of all
 *                 facts that can possibly be derived.
 */
private void deriveAll()
{









}
```

7. **(8 pts)** Write the method `updatePosse`. This method uses the `getImpossibilities` method to get a set of `Possibility` objects; these objects must be removed, if present, from the possibilities set for the given potion. For example, say we have the following:

```
potions:       [<tall green potion>,
                <medium blue potion>,
                <short yellow potion>]

potionsPosse:
     [<tall green potion> → [poison, proceed],
      <medium blue potion> → [poison, return],
      <short yellow potion> → [poison, wine, proceed]]
```

Now, we call `updatePosse` with an `index` of 2. So, we are looking at the `<short yellow potion>`. The `getImpossibilities` method returns `[poison, proceed, return]`. This means that the `<short yellow potion>` cannot be poison, it cannot help us proceed, and it cannot help us return – those were the impossibilities. So, we need to remove these possibilities from the `potionsPosse`. Thus, our final `potionsPosse` should look like this:

```
potionsPosse:
     [<tall green potion> → [poison, proceed],
      <medium blue potion> → [poison, return],
      <short yellow potion> → [wine]]
```

Note that `getImpossibilities` returned a set including `return`, even though `return` was not in the possibility set. That is OK – `getImpossibilities` may return possibilities that we have already canceled out.

```
/* Updates the possibility sets for a potion.
 * preconditions:  The facts field is set up;
 *                 0 <= index < potions.size()
 * postconditions: The potionsPosse field is updated
 *                 so that it does not have any
 *                 impossibilities for the potion at
 *                 the given index.  It is impossible
 *                 for this method to increase the
 *                 size of a possibility set.
 */
private void updatePosse(int index)
{
















}
```

8. **(4 pts)** Write the method `updateAllPosse`. This method uses `updatePosse` to update each potion's possibility set.

```
/* Updates all the possibility sets.
 * postconditions: The possibility set is updated for
 *                 every potion.
 */
private void updateAllPosse()
{











}
```

9. **(16 pts)** Write `findGoodPotion`. This method uses a queue of facts to build up the known facts and eliminate possibilities. It should use as few facts as possible to determine which potion we should use to proceed. For every fact in the queue, this method first adds it to the set of known facts and then uses other methods to derive all other possible facts. Once all the facts have been derived, we use our known facts to eliminate possibilities for the identities of each of the potions. Then, we check to see if any potion is guaranteed to be the potion used to proceed – this will happen when a potion has only one possibility remaining, and that possibility indicates it's the potion used to proceed. If we find our potion, return it; otherwise, get the next fact and try again. This method returns `null` if the queue of facts is exhausted and if it still has not determined the correct potion without ambiguity.

```
/* Uses a Queue of facts to eliminate possibilities,
 * in the hopes of finding the potion that moves us
 * forward.
 * preconditions:  queue is a valid queue of facts;
 *                 all fields are initialized properly
 * postconditions: returns the Potion that will allow
 *                 us to proceed if we can identify
 *                 the potion uniquely from the given
 *                 facts; otherwise, returns null.
 */
public Potion findGoodPotion(Queue<Fact> queue)
{




}
```

After applying all the facts at their disposal, Harry and Hermione discover that the potion used to proceed has but only one small sip left. The two agree that Harry should continue alone; Hermione will take a different potion to return to find Ron, who was last seen sacrificing himself on the chess board.

Harry takes the potion, turning his insides to ice, and walks through the black fire.

The room he enters is different than the ones that came before – it is larger and well lit. After his eyes adjust, Harry sees an odd sight. Professor Quirrell, Harry's Defense Against the Dark Arts teacher, is sitting in front of a computer terminal. From the look of things, the Professor is just about ready to break the machine out of frustration. Unaware of Harry's presence, he mutters to himself, "I found the Factorer's Stone – that was easy – but now I can't get it to work!"

Hearing footsteps behind him, Quirrell rounds upon Harry. Harry, known as the best computer science student at Hogwarts, is quickly threatened with dismemberment and death unless he can figure out how to use the Factorer's Stone.

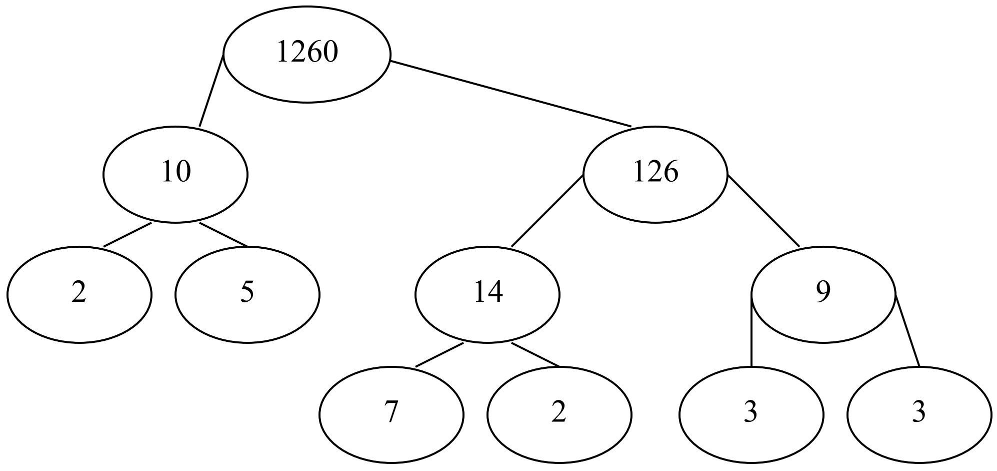
The Stone provides the following class:

```
public class Stone
{
     /* Returns one factor of a number in constant time.
      * preconditions:  product > 1
      * postconditions: if product has a factor greater
      *                 than 1 (in other words, if product
      *                 is not prime), returns a factor
      *                 greater than one.  Otherwise,
      *                 returns -1.  This method will
      *                 return the same factor for a given
      *                 number every time.  For example,
      *                 getFactor(1260) will always be 10.
      *                 To get the matching factor, use
      *                 division.
      */
     public static int getFactor(int product)
     { /* implementation is magical */ }
}
```

As all Hogwarts computer scientists know, being able to factor large numbers quickly allows you essentially to control the world. This power is what Quirrell is after.

In order to use the Stone's ability to factor, we must build a binary tree of factors. Every node in this binary tree (which is *not* a binary search tree) contains an `Integer`. The root of the tree is the number we are trying to factor. Its two children are factors of that number. The children that are not prime have two children, both of which are factors of the child. This continues down the tree until all leaves are prime.

For example, consider the following tree:



This is a valid factor tree. Note that there is no particular ordering to the two children and that there are many valid factor trees rooted with the number 1260.

10. **(4 pts)** In what order are the nodes of the above tree visited in a preorder traversal?

    A. 2, 2, 3, 3, 5, 7, 9, 10, 14, 126, 1260

    B. 2, 10, 5, 1260, 126, 14, 7, 2, 9, 3, 3

    C. 2, 10, 5, 1260, 7, 14, 2, 126, 3, 9, 3

    D. 1260, 10, 2, 5, 126, 14, 7, 2, 9, 3, 3

    E. 1260, 10, 2, 5, 126, 9, 3, 3, 14, 2, 7

11. **(4 pts)** In what order are the nodes of the above tree visited in an inorder traversal?

    A. 2, 2, 3, 3, 5, 7, 9, 10, 14, 126, 1260

    B. 2, 10, 5, 1260, 126, 14, 7, 2, 9, 3, 3

    C. 2, 10, 5, 1260, 7, 14, 2, 126, 3, 9, 3

    D. 1260, 10, 2, 5, 126, 14, 7, 2, 9, 3, 3

    E. 1260, 10, 2, 5, 126, 9, 3, 3, 14, 2, 7

Here is the definition of the `FactorTree` class, using the `Node` from *Node.java* as posted on class 20 of CS206:

```
public class FactorTree
{
     // the root of the tree
     private Node<Integer> root;

     /* Constructs a tree rooted with a given number
      * preconditions:  num > 1
      * postconditions: root refers to the root of num's
      *                 factor tree
      */
     public FactorTree(int num)
     {
          root = makeTree(num);
     }

     /* Creates a tree rooted with a given number
      * preconditions:  num > 1
      * postconditions: returns the root of num's factor tree
      */
     private Node<Integer> makeTree(int num)
     { /* to be implemented below */ }

     /* Returns a List of prime factors for this tree
      * preconditions:  tree is built
      * postconditions: a List containing all the prime
      *                 factors is returned.
      */
     public List<Integer> getPrimeFactors()
     {
          List<Integer> retVal = new LinkedList<Integer>();
          primeFactorsHelper(root, retVal);
          return retVal;
     }

     /* Adds the prime factors in one subtree to a list
      * preconditions:  tree is a valid factor subtree
      * postconditions: all the prime factors in the
      *                 subtree rooted at 'tree' are added
      *                 to the list, in any order.
      */
     private void primeFactorsHelper(Node<Integer> tree,
                                     List<Integer> list)
     { /* to be implemented below */ }
}
```

12. **(12 pts)** Write the method `makeTree`. This method builds and returns a factor tree with the given number as its root value. It is recursive.

```
/* Creates a tree rooted with a given number
 * preconditions:  num > 1
 * postconditions: returns the root of num's factor tree
 */
private Node<Integer> makeTree(int num)
{









}
```

13. **(12 pts)** Write the method `primeFactorsHelper`. It appends all the prime factors from a subtree of the factor tree to a list. It, too, is recursive.

```
/* Adds the prime factors in one subtree to a list
 * preconditions:  tree is a valid factor subtree
 * postconditions: all the prime factors in the
 *                 subtree rooted at 'tree' are added
 *                 to the list, in any order.
 */
private void primeFactorsHelper(Node<Integer> tree,
                                List<Integer> list)
{








}
```

Harry diligently works on writing `FactorTree`, knowing that with every keystroke, he is getting closer to granting the corrupt Quirrell unimaginable power. Just as he finishes the last method, we are saved! When Hermione revived Ron, they both escaped to find the headmaster of Hogwarts, Professor Albus Dumbledore. He saved Harry just in time and destroyed the Factorer's Stone. The Ministry of Magic will no longer be able to read wizards' encrypted messages, but the world is safe from those who would use the Stone for evil instead of good.

THE END

Extra credit problem:

14. (**20 extra-credit pts, for an impeccable solution)**[2]

Here is the riddle of the potions:

Danger lies before you, while safety lies behind,
Two of us will help you, whichever you would find,
One among us seven will let you move ahead,
Another will transport the drinker back instead,
Two among our number hold only nettle wine,
Three of us are killers, waiting hidden in line.
Choose, unless you wish to stay here forevermore,
To help you in your choice, we give you these clues four:
First, however slyly the poison tries to hide
You will always find some on nettle wine's left side;
Second, different are those who stand at either end,
But if you would move onward, neither is your friend;
Third, as you see clearly, all are different size,
Neither dwarf nor giant holds death in their insides;
Fourth, the second left and the second on the right
Are twins once you taste them, though different at first sight.[3]

Here are the bottles:

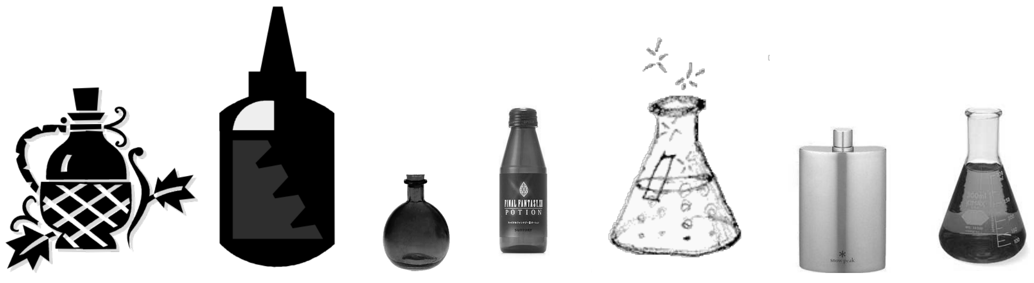


Figure out both which bottle is the one that allows you to proceed and the one that allows you to go back. Explain how you arrived at your answer.

[2] This problem is inspired by the riddle in the book (*Harry Potter and the Sorcerer's Stone*, p. 285) and work done by "Prefect Marcus" and presented on http://www.hp-lexicon.org/essays/essay-potionriddle.html
[3] *Harry Potter and the Sorcerer's Stone*, p. 285.